Diplomarbeit

Konzeption und Implementierung
eines dreidimensionalen
Klassenbrowsers für Java

Frank Engelen



Diplomarbeit am
Lehrstuhl für Softwaretechnologie
Fachbereich Informatik
Universität Dortmund

Gutachter:
Prof. Dr. E.-E. Doberkat
Dipl.-Inform. K. Alfert

1. September 2000

Zusammenfassung

In der Softwareentwicklung ist der Einsatz von Diagrammen schon lange üblich. Die Komplexität moderner Softwaresysteme führt allerdings dazu, daß Diagramme, die deren Struktur verdeutlichen sollen, oft unübersichtlich erscheinen. Dies gilt insbesondere bei einer automatischer Generierung der Darstellung. Verschiedene Studien haben gezeigt, daß sich durch dreidimensionale Graphiken umfangreiche Sachverhalte besser verdeutlichen lassen [WF96, SM93]. Ziel dieser Arbeit ist es deshalb, ein Konzept für eine dreidimensionale Visualisierung von Softwarestrukturen aufzuzeigen und dieses prototypisch umzusetzen. Dies erfolgt am Beispiel von Software, die in der Programmiersprache Java implementiert ist.

Das entwickelte Konzept beruht insbesondere auf einer Kombination verschiedener zwei- und dreidimensionaler Visualisierungstechniken, wie z.B. der Cone Trees [RMC91] oder Information Cubes [Rek93]. Es zeigte sich, daß Sprachkonzepte von Java, wie z.B. Klassifizierung oder Vererbung, nicht starr auf bestimmte Darstellungstechniken abgebildet werden sollten, sondern eine Lösung vorzuziehen ist, die es erlaubt, zwischen verschiedenen Techniken gemäß der Eigenschaften des zu zeigenden Sachverhaltes auszuwählen. So entstand eine Art „Baukasten für Visualisierungen", der Anwendern eine Reihe von Techniken anbietet, die sie flexibel einsetzen können.

Für die Umsetzung des Konzepts wurde eine Kombination aus einem Java-Applet und der *Virtual Reality Modeling Language* (VRML) [ANM97] eingesetzt.

Danksagung

An dieser Stelle möchte ich mich aufrichtig bei Dipl.-Inform. Klaus Alfert und Dipl.-Inform. Alexander Fronk für ihre geduldige Betreuung bedanken. Durch ihre freundliche Unterstützung und ihre stets konstruktive Kritik haben sie mir viele Denkanregungen geliefert, die mir bei der Arbeit sehr geholfen haben.

Claudia Hellinger danke ich für die Durchsicht der Arbeit, Frank Kloster für das Bereitstellen seines Computers zu Testzwecken.

Dafür, daß sie mir mein Studium überhaupt erst ermöglicht haben, gebührt meinen Eltern besonderer Dank, der um so größer ausfallen muß, da sie mich während der oft stressigen Zeit vor Prüfungen und Abgaben jederzeit moralisch unterstützt haben. Ganz wesentlich hat mir in dieser Zeit auch Petra Lamparski geholfen. Für ihre Liebe und Geduld möchte ich mich bei ihr herzlich bedanken. Endlich können wir sagen: „*Geschafft!*".

Inhaltsverzeichnis

Teil IV – Abschluß

Anhänge

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

3D	dreidimensional
API	Application Programming Interface
AWT	Abstract Window Toolkit
BALSA	Brown University Algorithm Simulator and Animator
CAD	Computer Aided Design
DOI	Degree of Interest
EAI	External Authoring Interface
FAQ	Frequently Asked Questions
FEV	Fish Eye View
FSN	File System Navigator
GV3D	GraphVision3D
HCI	Human Computer Interaction
HTML	Hypertext Markup Language
JDK	Java Development Kit
LOD	Level of Detail
NV3D	NestedVision3D
RCS	Resource Control System
UML	Unified Modeling Language
VQN	Vollständig qualifizierender Name
VRCS	Visual Resource Control System
VRML	Virtual Reality Modeling Language
VS	Visualisierungssystem
VT	Visualisierungstechnik
WWW	World Wide Web

Teil I

Einführung

1 Einleitung

Der Einsatz von Computertechnik ist heutzutage vielfach mit sehr hohen Erwartungen verbunden. Es gilt den Geschäftsbetrieb ganzer Konzerne zu unterstützen oder komplizierte industrielle Anlagen zu steuern. Immer größere Anforderungen insbesondere auch an die eingesetzte Software führen dazu, daß deren Entwicklung zunehmend komplexer wird. Eine Unterstützung der Softwareproduktion durch Werkzeuge ist daher unabdingbar geworden.

Während der Implementierung und Wartung objektorientierter Software sind Klassenbrowser ein wichtiges Hilfsmittel. Durch sie wird es Softwareentwicklern erleichtert, Informationen über die Gliederung der betrachteten Software zu gewinnen. Zur Veranschaulichung dieser Informationen werden vielfach graphische Darstellungen (sog. *Visualisierungen*) verwendet, da davon ausgegangen wird, daß diese, verglichen mit rein textuellen Beschreibungen, für einen Betrachter leichter und vor allem schneller verständlich sind.

Mit steigender Komplexität der darzustellenden Information kommt es dazu, daß auch ihre graphische Darstellung nicht mehr als Ganzes überblickt werden kann und somit die erwartete Verständlichkeit u.U. verloren geht. Verschiedene Studien haben gezeigt, daß sich durch dreidimensionale Graphiken umfangreiche Sachverhalte besser verdeutlichen lassen [WF96, SM93]. Ziel diese Arbeit ist es deshalb ein Konzept aufzuzeigen, mit dem sich die Struktur objektorientierter Software geeignet dreidimensional visualisieren läßt. Dies erfolgt am Beispiel von Software, die in der Programmiersprache Java implementierter ist (im folgenden kurz *Java-Software*) [AG98]. Das gezeigte Konzept wird in einem dreidimensionalen Klassenbrowser prototypisch umgesetzt.

Die Arbeit ist in vier Teile gegliedert. Der Rest des ersten Teils befaßt sich zunächst im Kapitel 2 mit der Motivation für diese Arbeit, wobei besonders auf mögliche Vorteile dreidimensionaler Darstellungen eingegangen wird. Im dritten Kapitel werden dann einige Grundlagen zu dieser Arbeit vermittelt. Dazu gehören Begriffe der dreidimensionalen Computergraphik sowie eine nähere Betrachtung des Visualisierungsbegriffs.

Der sich anschließende zweite Teil beschäftigt sich mit der Konzeption einer dreidimensionalen Darstellungsform für den Aufbau von Java-Software.

Der dritte Teil beschreibt die prototypische Implementierung eines auf der konzipierten Darstellungsform basierenden Klassenbrowsers.

Die Arbeit schließt mit einem vierten Teil, der eine Bewertung des Erreichten sowie einen Ausblick auf weitere Entwicklungsmöglichkeiten liefert.

2 Motivation

In diesem Kapitel werden vier Fragen betrachtet. Zunächst wird aufgezeigt, warum für diese Arbeit Java als Grundlage gewählt wurde (Abschnitt 2.1). Dann wird kurz darauf eingegangen, warum man Visualisierungen überhaupt verwendet (2.2). Danach werden – etwas ausführlicher – Probleme beschrieben, die bei Visualisierungen auftreten können (2.3). Vor allem wird in diesem Kapitel aber untersucht, warum man hoffen darf, daß dreidimensionale Visualisierungen einige dieser Probleme lindern (2.4).

2.1 Warum Java?

Die Vorstellung der Programmiersprache Java Im Jahr 1995 durch die Firma Sun war Ausgangspunkt einer Welle von verschiedenartigen publizierten Beiträgen zur Sprache, auf den Markt gebrachten Entwicklungsumgebungen und auch einer Reihe sonstiger Werkzeuge. Innerhalb kurzer Zeit gelangte Java zu einem hohen Bekanntheitsgrad. Die Sprache läßt sich durch die folgenden Eigenschaften charakterisieren [Küh96]:

- *objektorientiert*,
- *einfach*, z.B. durch die Beschränkung auf wenige Konstrukte,
- *robust*, z.B. durch eine strenge Typisierung und den Verzicht auf Zeigerarithmetik,
- *plattformunabhängig*, durch einen interpretativen Ansatz und eine umfangreiche und standardisierte Klassenbibliothek,
- *sicher* in dem Sinne, als daß die Handlungsmöglichkeiten eines Java-Programmes durch den Anwender des Programmes beschränkt werden können und dieser es so z.B. verhindern kann, daß Programme unbekannten Ursprungs auf die Festplatte seines Rechners zugreifen.

Dagegen sprechen kritische Stimmen häufig vom Performanzproblemen, die mit der Plattformunabhängigkeit einher gehen. Ein weiterer Kritikpunkt ist z.B. das Fehlen generischer Klassen. Nichtsdestoweniger scheint sich Java – nachdem die Sprache längere Zeit in dem Ruf stand, nur für kleine Applikationen geeignet zu sein, und hier insbesondere für solche, die im Zusammenhang zum bekanntem World Wide Web (WWW) stehen – zunehmend auch in Bereichen vorzudringen, die bis dato traditionellen Programmiersprachen vorbehalten waren, wie z.B. das Hochleistungsrechnen [Phi00]. Mit immer weiter gefaßten Einsatzgebieten wird sich auch hier der Wunsch nach einer Werkzeugunterstützung verstärken, die über das bis dato gebotene hinausgeht. Insbesondere sind umfangreichere Java-Programme zu erwarten, zu deren Visualisierung neue Wege gefunden werden müssen.

Aber nicht nur die Popularität spricht für die Wahl von Java als Grundlage dieser Arbeit. Java bietet zudem eine Vielzahl von Konzepten wie z.B. Klassen, Schnittstellen, Pakete, Vererbung, Implementierung usw. (vgl. Kapitel 5) deren Visualisierung lohnend erscheint, da durch ihre Anwendung der Aufbau eines Softwaresystems geprägt wird. Einige dieser Konzepte finden sich auch in anderen objektorientierten Programmiersprachen, wie z.B. C++ [ES90] wieder, so daß sich Erkenntnisse, die für Java gewonnen wurden, übertragen lassen dürften. Gegen die Wahl von C++ sprach die dort vorliegende Vermischung von prozeduralen und objektorientierten Konzepten.

2.2 Warum Visualisierung?

Die Motivation für Visualisierungen ist ein Gedanke, der sich gut durch die Redensart *„ein Bild sagt mehr als tausend Worte*" ausdrücken läßt. Komplexe Sachverhalte lassen sich nur schwer als ganzes überblicken. Textuelle Beschreibungen geraten oft detailliert und

umfangreich, so daß hier der Überblick schwer erreicht wird und schnell wieder verloren gehen kann. Bilder können von Menschen i.d.R. leichter rezipiert werden als Text. Ein Grund hierfür dürfte sein, daß das Lesen von Text ein sequentieller Vorgang ist. Das Betrachten eines Bildes hingegen erfolgt normalerweise nicht sequentiell von oben nach unten und von links nach rechts. Die stattfindende parallele Betrachtung unterschiedlicher Bildteile entspricht eher den kognitiven Fähigkeiten des Menschen.

Die Idee, Sachverhalte graphisch aufzubereiten, ist auch innerhalb der Informatik bereits seit langem etabliert. Früh wurden z.B. Programmablaufpläne eingesetzt, die den Ablauf von Programmen graphisch beschreiben [Eng93, S.542f]. Im Laufe der Zeit sind für unterschiedliche Problemstellungen innerhalb der Informatik eine Reihe von graphischen Sprachen entwickelt worden, um die entsprechenden Probleme oder deren Lösung darzustellen. So entstand z.B. für das Problem der Datenmodellierung die bekannten Entity/Relationship-Diagramme (ER-Diagramme) [Eng93, S.232ff]. Weit verbreitet sind auch Petrinetze [Eng93, S.520ff] oder Klassendiagramme für objektorientierte Software in verschiedenen Notationen wie z.B. der Booch-Notation [Boo94] oder der Unified Modeling Language [BRJ99].

2.3 Probleme bei Visualisierungen

Visualisierungen sind allerdings kein Allheilmittel. Zu dieser Erkenntnis kommen z.B. Larkin und Simon in einem Artikel, der, in Anlehnung an die oben zitiere Redewendung, mit „*Why a Diagram is (Sometimes) Worth Then Thousand Words*“ überschrieben ist [LS87]. Die in der Abbildung 2.1 dargestellte Karikatur läßt einige der Probleme erahnen, die mit Visualisierungen verbunden sein können. Zu diesen zählen:

- Unklare Bedeutung der verwendeten Symbole.
- Unstrukturiertheit der Darstellung.
- Überfrachtung der Darstellung.

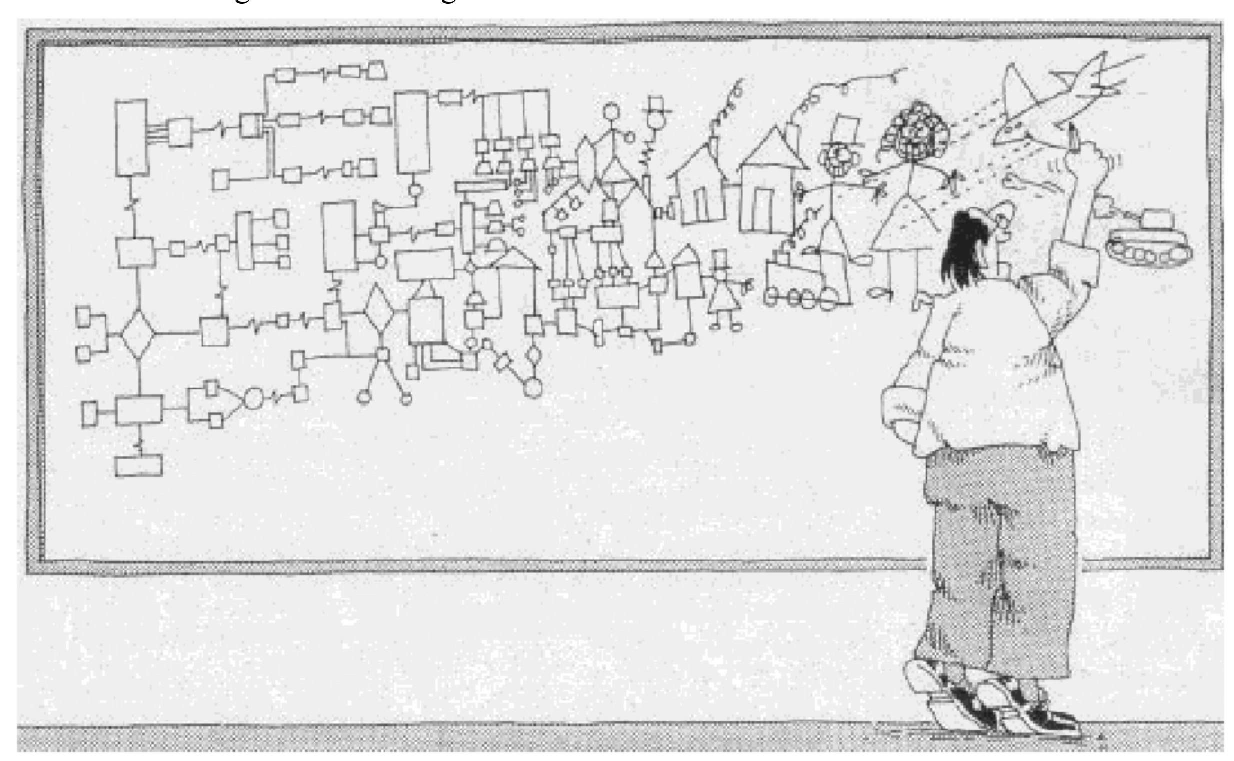
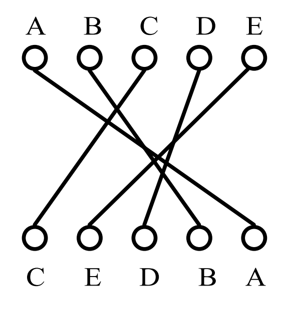


Abbildung 2.1: Karikatur zur Visualisierung (Quelle: [Ten94] nach [Eng95])

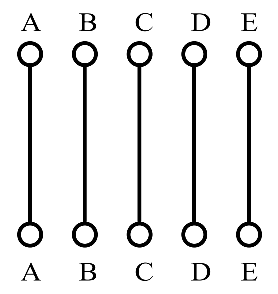
Eine Darstellung wird aus einzelnen Symbolen aufgebaut. Wenn deren Bedeutung nicht

festgelegt ist, kann auch die Bedeutung der gesamten Darstellung nicht erschlossen werden. Besonders problematisch wird dies, wenn mangels einer Festlegung ein Symbol mit unterschiedlicher Semantik verwendet wird. Zudem muß nach Möglichkeit aus der Gestalt eines Symbols intuitiv auf dessen Sinn geschlossen werden können, damit ein immer wiederkehrendes Nachdenken und ggf. ein Nachschlagen in Legenden vermieden wird.

Eine wahlfreie Anordnung von Symbolen in der Darstellung kann allerdings deren Aussagekraft trotz noch so intuitiver Symbole verderben. Ein besonderes Problem bei der Darstellung von Graphen ist, daß es schnell zu Überschneidungen von Kanten kommt. Diese erschweren das Verständnis (vgl. Abbildung 2.2). Im Extremfall kann es dazu kommen, daß ein Betrachter nur noch ein Gewirr aus Linien und Symbolen zu erkennen vermag.



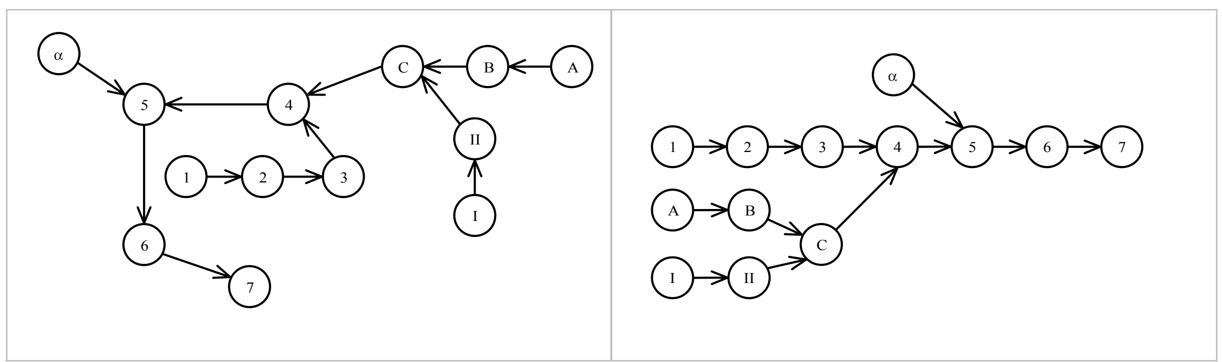
- a - mit Überschneidungen



- b - ohne Überschneidungen

Abbildung 2.2: Zweimal derselbe Sachverhalt: einmal dargestellt mit Überschneidungen und einmal ohne.

Das Vermeiden von Überschneidungen führt allein aber nicht immer zu einer guten Aufteilung der Darstellung. Vielmehr sollten sich Eigenschaften der Struktur des Sachverhalts auch in der Struktur seiner Darstellung wiederfinden. Wird ein Graph beispielsweise zur Beschreibung eines zeitlichen Ablauf von Aktivitäten eingesetzt, so bietet es sich an, die Darstellung des Graphen ebenfalls gemäß des Ablaufs zu strukturieren (vgl. Abbildung 2.3).



- a - verfehlte Anordnung - b - Strukturierung gemäß zeitlichem Ablauf

Abbildung 2.3: Strukturierte und unstrukturierte Darstellung eines Graphens mit Aktivitäten als Knoten und der vorgeschriebenen Reihenfolge als Kanten

Mit einer Unstrukturiertheit geht häufig das Problem einher, daß Pfeile zwischen Symbolen unverhältnismäßig lang werden. Start- und Endpunkt eines Pfeils können dann nicht mehr auf einen Blick erfaßt werden. Dies ist besonders schwerwiegend, wenn die Darstellung des Graphen mittels eines Computers erfolgt. Hier wird es aufgrund der begrenzten Monitorgröße und -auflösung nicht immer möglich sein, den gesamten Graphen gleichzeitig zu präsentieren, so daß dieser nur ausschnittsweise oder verkleinert dargestellt werden kann. Bei langen Pfeilen ist die Wahrscheinlichkeit höher, daß einer der beiden Endpunkte nicht im momentan dargestellten Ausschnitt zu sehen ist und dieser somit gewechselt werden muß, was für den Betrachter einen zusätzlichen Aufwand beinhaltet. Auch eine Verkleinerung der Darstellung

kann nicht immer Abhilfe schaffen, da dann u.U. wichtige Details nicht mehr erkennbar sind, und somit ein ständiger Wechsel zwischen einer verkleinerten Sicht und einer detaillierten Sicht erforderlich wird.

Das Problem der Unstrukturiertheit verstärkt sich noch, wenn Visualisierungen nicht durch einen Menschen entworfen werden, sondern automatisch durch einer Computer berechnet werden sollen. Gerade für umfangreiche Sachverhalte, über die Daten ggf. bereits in einer maschinell lesbaren Form vorliegen, wünscht man sich eine derartige Berechnung, da ein manuelles Vorgehen mit einem hohem Aufwand verbunden ist. Insbesondere für die Darstellung von Graphen existieren bereits seit längerem eine große Anzahl von Algorithmen (vgl. [BET+94]). Diese liefern teilweise erstaunlich gute Resultate. Als ein Beispiel zeigt Abbildung 2.4 eine mit Hilfe der Softwarebibliothek *GLT 2.2* von der *Tom Sawyer Software Inc.* erzeugte Darstellung eines Telekommunikationsnetzwerkes [Tol96]. Als Eingabe für diese Art von Algorithmen dient häufig eine einfache Beschreibung des darzustellenden Graphens anhand einer Menge von Knoten und einer Menge von Kanten zwischen diesen Knoten. Diese einfache Datenstruktur erlaubt es i.d.R. nicht, alle Aspekte eines Sachverhalts zu beschreiben. So kommt es, daß die Qualität automatisch generierter Darstellungen häufig nicht an sorgfältig von Hand gestaltete Visualisierungen heranreichen kann, da dem menschlichen Gestalter sein Hintergrundwissen über den Sachverhalt zur Verfügung steht, um diesen besser zu repräsentieren.

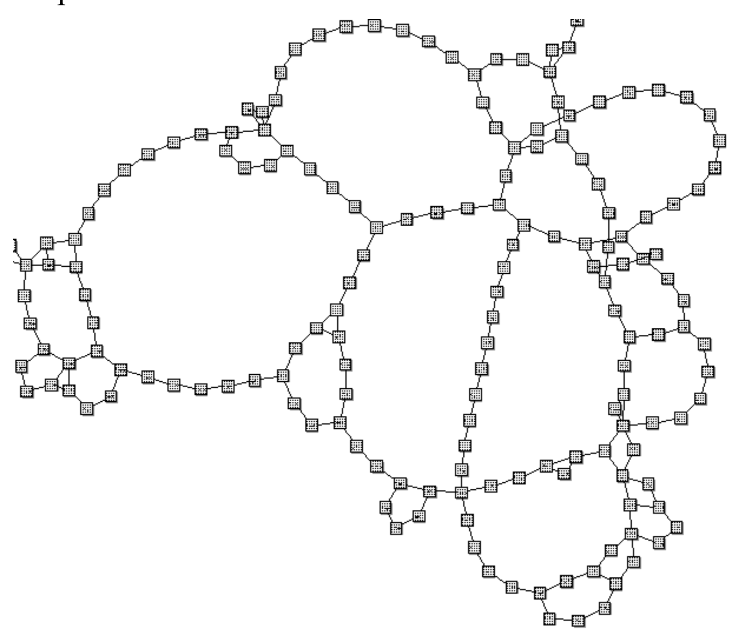


Abbildung 2.4: Automatisch erzeugte Darstellung eines Telekommunikationsnetzwerkes [Tol96]

Bei sehr komplexen Sachverhalten kann es zu wenig verständlichen Darstellungen kommen, wenn versucht wird zu viele – und vielleicht sogar vom Typ her unterschiedliche – Dinge in ein Bild zu fassen. Dieses erscheint dann mit lauter Details überfrachtet und ein Gesamtverständnis für den Sachverhalt ist nur schwer zu erlangen. Eine psychologische Studie hat gezeigt, daß es einem Menschen nur möglich ist, bis zu sieben plus/minus zwei Informationseinheiten gleichzeitig zu verarbeiten [MM56]. Daraus zu folgern, daß die Darstellung eines Graphen nur aus maximal neun für Informationseinheiten stehende Knoten und Kanten bestehen darf, erscheint aber zu pessimistisch, da nicht alle Bestandteile eines Graphens gleichzeitig betrachtet werden müssen. Ein Schwierigkeit ergibt sich aber z.B. dann, wenn ein Knoten über Kanten mit zu vielen anderen Knoten verbunden ist. Der Zusammenhang zwischen diesen Knoten kann dann nicht mehr „auf einen Blick" erfaßt werden.

2.4 Warum dreidimensionale Visualisierung?

Die genannten Probleme mit herkömmlichen – zweidimensionalen – Visualisierungen haben dazu geführt, daß verstärkt Möglichkeiten gesucht werden, dreidimensionale Computergraphik zur Qualitätsverbesserung zu nutzen. Genau wie Java erlebt auch die dreidimensionale Computergraphik zur Zeit einen Boom. Betrachtet man sich z.B. den aktuellen Katalog eines populären Softwarevertriebs, so stellt man fest, daß eine Vielzahl an Produkten die Bezeichnung "3D" (für dreidimensional) im Titel führen (vgl. z.B. [SMM00]). Zurückzuführen ist dies auf die immer weiter steigende Leistungsfähigkeit aktueller Computersysteme. Insbesondere zu nennen ist hier das Aufkommen sogenannter 3D-Graphikbeschleunigerkarten für die weit verbreiteten Personal Computer in den letzten Jahren. Diese Karten ermöglichen eine Verlagerung der teilweise sehr aufwendigen Berechnungen für dreidimensionale Graphiken von der Software zur Hardware, wodurch diese um ein vielfaches beschleunigt werden.

Aber das wachsende Interesse an dreidimensionalen Visualisierungen nur mit einer aktuellen Steigerung der verfügbaren Computerleistung zu begründen, greift zu kurz. Bereits vor mehr als zehn Jahren wurde im SemNet-Projekt versucht, durch die Anwendung dreidimensionaler Computergraphik die Effektivität von Visualisierungen großer Wissensdatenbanken zu verbessern [FPF88].

Wodurch begründet sich dann aber das Interesse an dreidimensionalen Visualisierungen? Wie kann der Einsatz dreidimensionaler Graphiken zur Milderung den oben genannten Problem beitragen bzw. welche Vorteile können erwartet werden? Darauf soll im folgenden ausführlicher eingegangen werden.

Ein besonders augenfälliger Vorteil ergibt sich bei der Darstellung mathematischer Funktionen in Koordinatensystemen. Hier steht eine Achse mehr zur Verfügung, die mit einer Bedeutung versehen werden kann. Abbildung 2.5 zeigt den durch eine Funktionen beschriebenen Zusammenhang zwischen einer Kraft und einem Ort. In der dreidimensionalen Darstellung kann zusätzlich noch der zeitliche Verlauf aufgetragen werden. Für den Betrachter ergibt sich so ein Gesamtbild, das er sonst erst durch die mühsamere Betrachtung und mentaler Integration mehrerer Graphiken, die den Kraft/Ort-Zusammenhang zu verschiedenen Zeitpunkten angeben, erlangt hätte.

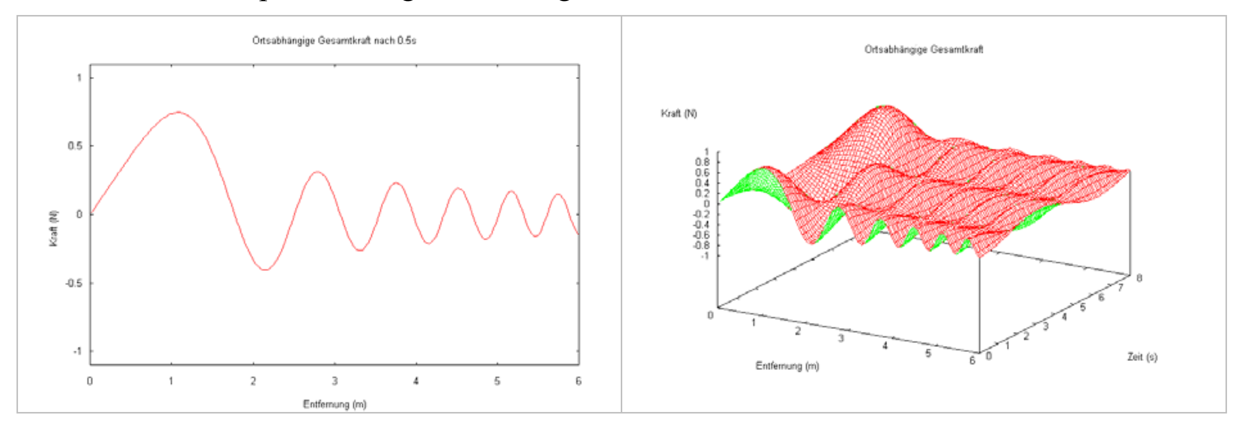
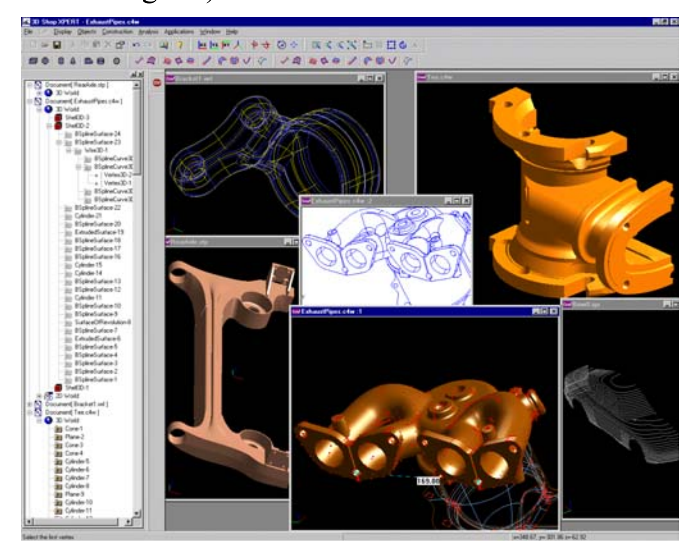


Abbildung 2.5: Zwei- und dreidimensionale Darstellung von Funktionen

Werden Erwartungen, die mit dem Einsatz der dreidimensionaler Visualisierung verbunden sind, genannt, so wird häufig auf deren „Natürlichkeit" verwiesen (vgl. z.B. [Wün97, S. 58]): für Menschen ist das Leben in dreidimensionalen Räumen tägliche Erfahrung. Somit fällt es ihnen leicht, sich im Raum zu lokalisieren. Bei dreidimensionalen Graphiken kommt es schnell zu dem Effekt, daß sich der Betrachter in die dargestellte Szene hineinversetzt fühlt und sich in dieser wie in einem natürlichen Raum lokalisieren kann, wodurch die Navigation

in der Szene begünstigt wird. Bei einer zweidimensionalen Abbildungen ist ein solches gedankliches Hineingehen nicht zu erwarten. Zudem hilft bei der Navigation in dreidimensionalen Darstellungen das räumliche Erinnerungsvermögen des Menschen.

Geeignete Graphiksoftware vorausgesetzt, ergibt sich mit der Möglichkeit, die Darstellung aus verschiedenen Blickwinkeln zu betrachten und sie, gleichsam eines Werkstückes in der Hand, zu drehen und zu untersuchen, schnell ein intensiverer Eindruck vom dargestellten Sachverhalt – ein Umstand, der in konstruktiven Ingenieurdisziplinen, wie z.B. dem Maschinenbau, im Rahmen des Computer Aided Designs (CAD) besonders genutzt wird (vgl. Abbildung 2.6).



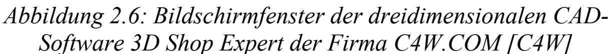


Abbildung 2.6: Bildschirmfenster der dreidimensionalen CAD-Software 3D Shop Expert der Firma C4W.COM [C4W]

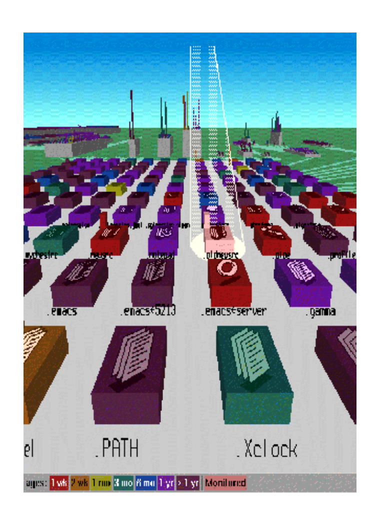


Abbildung 2.7: Beispiel für die Nutzung der Perspektive [Quelle:You96]

Bei der Visualisierung sehr komplexer Sachverhalte kann die bei dreidimensionaler Graphik mögliche perspektivische Verzerrung verwendet werden, um das Problem der Überfrachtung zu mildern. Teile des Sachverhalts, die man im Moment genauer betrachten möchte, werden dann im Vordergrund dargestellt, wodurch ggf. auch Details sichtbar werden. Es ist für den Betrachter aber auch wichtig, die Einbettung dieser Details in den Gesamtzusammenhang weiter vor Augen zu haben. Die Darstellung dieses Gesamtzusammenhanges kann im Hintergrund erfolgen, so daß dieser perspektivisch verzerrt und damit kleiner wird. Durch diese Verkleinerung wird es möglich, gleichzeitig mehr vom Kontext darzustellen, wodurch z.B. die Orientierung in der Darstellung erleichtert wird. In Abbildung 2.7 wird die Perspektive zur Darstellung von Datei in einem Dateisystem genutzt, wobei im Fokus der Betrachtung liegende Dateien im Vordergrund sind. Auch in diesem Zusammenhang kann wieder auf die Natürlichkeit der sich ergebenden Darstellung verwiesen werden, da ein Wechsel des aktuell fokussierten Darstellungsausschnitts durch eine Navigation auf den Hintergrund hin erfolgen kann – gleichsam des näher Herantreten an einen interessanten Gegenstand in der realen Welt.

Vielleicht aufgrund dieser Natürlichkeit kommt es dazu, daß dreidimensionale Darstellungen dem Betrachter attraktiver erscheinen als „trockene“ zweidimensionale Diagramme. Sie laden viel mehr zu einem explorartivem oder gar spielerischem Durchwandern ein, was dazu führen kann, daß sich motivierter und damit intensiver mit einem Sachverhalt beschäftigt wird und so ein tieferes Verständnis erlangt werden kann.

Oben wurden als zwei mögliche Probleme bei der Darstellung von Graphen sich überschneidende und zu lange Kanten genannt. Für das erste Problem bietet der Einsatz dreidimensionaler Graphiken theoretisch eine vollständig Lösung, da nach Xiam und Milgram jeder Graph hier überschneidungsfrei darstellbar ist [XM92]. Auch für das Problem der

langen Kanten kann man eine Milderung erwarten, denn es ist um jedes Symbol herum mehr Platz vorhanden, um verbundene Symbole in dessen Nähe zu plazieren. Abbildung 2.8 zeigt wie bei gleicher gewünschter Entfernung zwischen den Symbolen mehr Symbole um einen Mittelpunkt plaziert werden können.

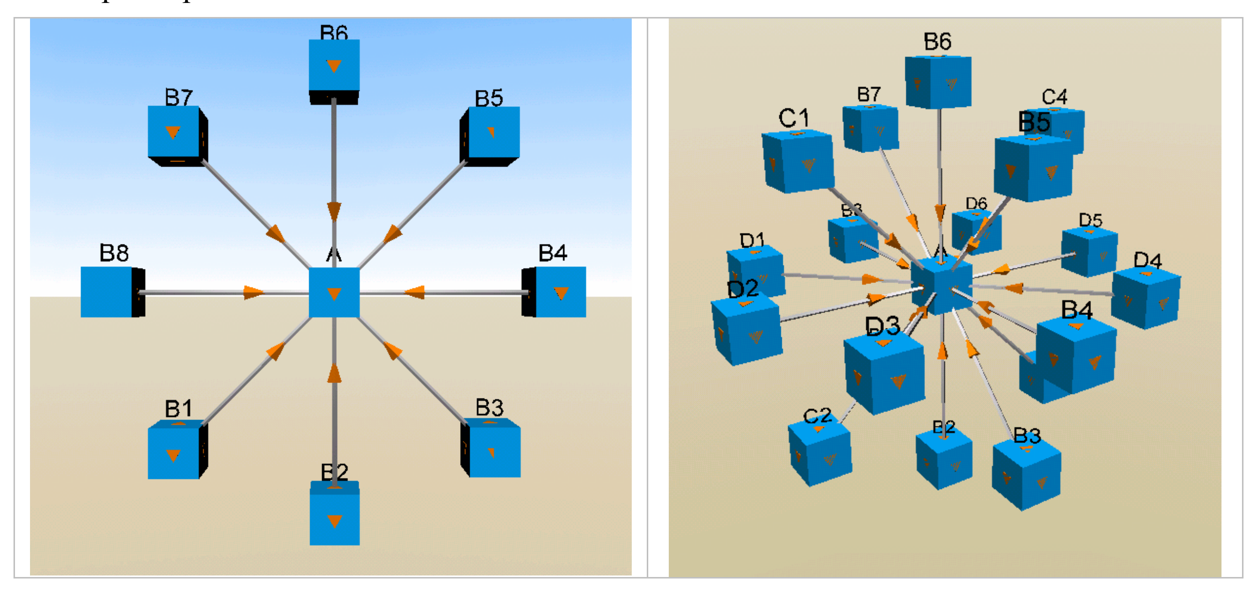


Abbildung 2.8: Plazierung von verbundenen Symbolen im Zwei- und Dreidimensionalen

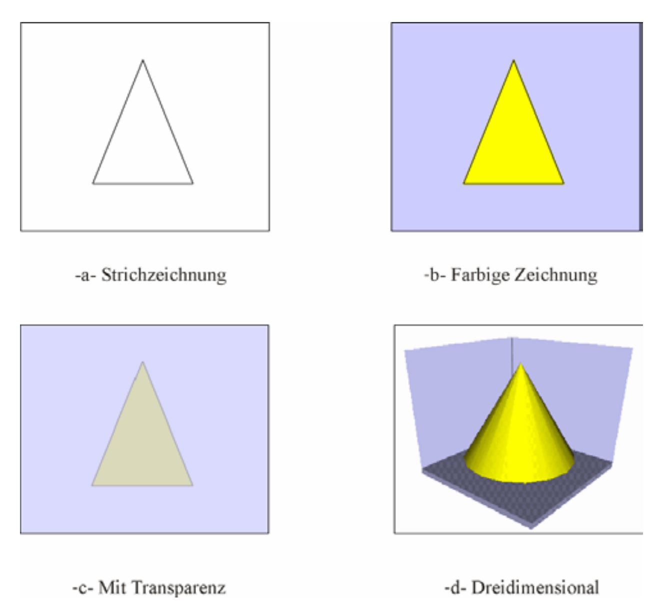


Abbildung 2.9: Darstellung der Zugehörigkeit eines Teils zu einem Ganzen (erweitert nach [AF00])

Häufig will man die Zugehörigkeit eines Teils zu einem Ganzen darstellen, in der Softwaretechnologie bspw. die Zugehörigkeit eines Subsystems zum Gesamtsystem. Oft wird dazu die Schachtelung von Symbolen verwendet (vgl. Abbildung 2.9). Dies ist bei zweidimensionalen Zeichnungen insbesondere bei gleichzeitiger Verwendung von Farben problematisch (vgl. auch [AF00], wo sich eine vergleichbare Betrachtung für unkolorierte Darstellungen finden läßt). Der Einsatz von Farben kann aber zur Visualisierung weiterer Eigenschaften sinnvoll sein. Während bei zweidimensionalen Strichzeichnungen wie in Abbildung 2.9a – vielleicht aus Gewöhnung an derartige Darstellungen – die Zugehörigkeit noch recht gut hervortritt, wirft Abbildung 2.9b die Frage auf, ob das Dreieck nicht zum Rechteck gehört, sondern vor diesem liegt. Durch die Verwendung semitransparenter Farben

kann der Eindruck etwas verbessert werden, hier kann sich obige Frage aber auch umdrehen, so daß jetzt vermutet werden kann, daß das Dreieck hinter dem Rechteck liegt (Abbildung 2.9c). In dreidimensionalen Schachtelungen wird die Zugehörigkeit bei weitem deutlicher (Abbildung 2.9d).

Werden Symbole entworfen, wird oft die Forderung nach einer leichten Skizzierbarkeit per Hand erhoben. Beispielsweise wurde die Booch-Notation [Boo94] für den objektorientieren Entwurf dahingehend kritisiert, daß das für Klassen verwendete Wolkensymbol (vgl. Abbildung 2.10) nicht leicht von Hand zu zeichnen sei. Daher sind Symbole oft vergleichsweise einfach gehalten. Bei dreidimensionalen Visualisierungen ist die Forderung nach Skizzierbarkeit hinfällig – man wird immer auf eine Rechnerunterstützung angewiesen sein. Dies könnte sich als ein Vorteil erweisen, da sich hierdurch die Möglichkeit ergibt, komplexere und damit vielleicht intuitivere Symbole zu verwenden. Weiterhin stehen mit der Rechnerunterstützung zusätzliche Möglichkeiten wie Animation oder Farbgebung zur Verfügung, die auf dem Papier nicht oder nur schwer nachvollziehbar sind.

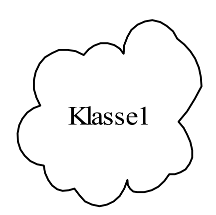
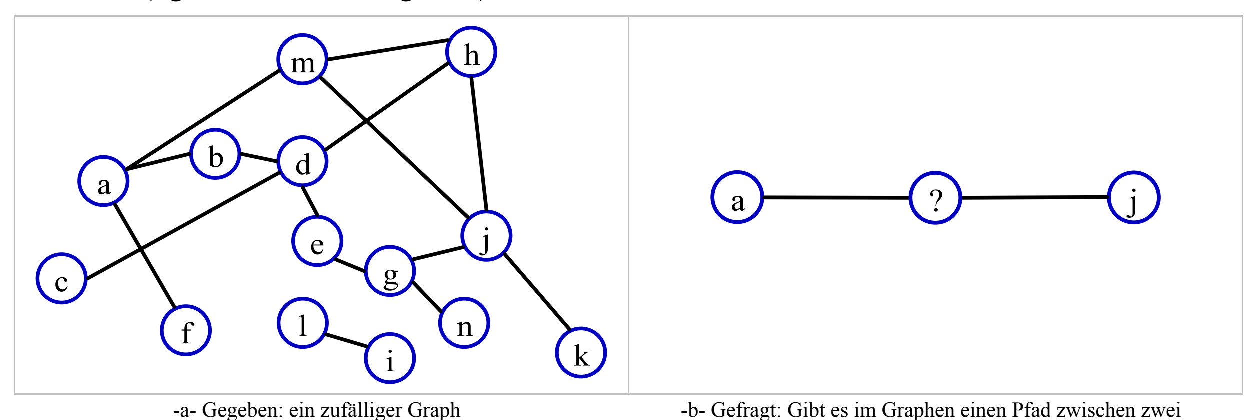


Abbildung 2.10: Symbol für Klassen in der Booch-Notation

Die prinzipielle Eignung dreidimensionaler Visualisierungen wurde durch verschiedene Studien gezeigt. Zum Beispiel vergleichen Ware und Franck in einer ihrer Arbeiten die zwei- und dreidimensionale Visualisierung von Graphen [WF96]. Die Knoten eines Graphen wurden dazu zufällig in einer Darstellungsebene bzw. in einem Darstellungsraum plaziert. Weiterhin waren verschiedene Knoten – ebenfalls zufällig bestimmt – paarweise miteinander verbunden. Einer Reihe von Probanden wurde nun aufgetragen festzustellen, ob zwischen zwei bestimmten Knoten eine Verbindung derart vorlag, daß vom ersten bestimmten Knoten eine Verbindung zu einem beliebigen Knoten und von diesem dann weiter zum zweiten vorgegebenen Knoten bestand. Man spricht in diesem Zusammenhang vom *Path Tracking Problem* (vgl. auch Abbildung 2.11).



-a- Gegeben: ein zufälliger Graph

-b- Gefragt: Gibt es im Graphen einen Pfad zwischen zwei bestimmten Knoten über einen beliebigen Zwischenknoten?

Abbildung 2.11: Illustration des Path Tracking Problems

Es wurde sowohl die Zeit, die ein Proband zur Lösung des Path Tracking Problems brauchte, als auch die Anzahl falscher Antworten gemessen. Dabei stellte sich dreierlei heraus:

- Bei der zweidimensionalen Visualisierung ist die Fehlerrate bis zu dreimal höher als bei der dreidimensionalen Visualisierung.
- Die Fehlerrate wird bei dreidimensionalen Visualisierungen maßgeblich durch Maßnahmen zur Unterstützung des Eindrucks von der Tiefe der Darstellung beeinflußt.
- Die benötigte Zeit ist nahezu unabhängig von der eingesetzten Visualisierungsform.

Eine Maßnahme zur Unterstützung des Tiefeneindrucks ist z.B. die Verwendung sogenannter Eye-Shutter-Brillen. Aufgrund der Wichtigkeit derartiger Maßnahmen, die auch durch eine Studie von Sollenberger und Milgram [SM93] bestätigt wird, werden sie im Abschnitt 3.3 nochmals gesondert behandelt.

Die Vielzahl der hier aufgezählte Argumente für dreidimensionale Visualisierungen läßt es lohnend erscheinen, diese auch für die Visualisierung von Softwarestrukturen anzuwenden. Diese Arbeit soll mithelfen, die Frage zu klären, wie eine solche Anwendung zweckmäßig gestaltet und umgesetzt werden kann.

3 Grundlagen

In diesem Kapitel wird zunächst der Visualisierungsbegriff und sein Umfeld beleuchtet (Abschnitt 3.1), dann werden drei eigene Termini definiert (3.2) und schließlich wird auf einige Grundlagen der dreidimensionalen Computergraphik eingegangen (3.3).

3.1 Der Visualisierungsbegriff und sein Umfeld

Die Begriffe „Visualisierung“ und auch „visualisieren“ wurden mit ihrem umgangssprachlichen Sinn bereits mehrfach verwendet. In diesem Abschnitt soll deren Bedeutung etwas genauer gefaßt werden. Weiterhin werden Termini eingeführt, die häufig in der einschlägigen Literatur verwendet werden.

Das Verb „visualisieren“ kann mit „*für das Auge gefällig gestalten*“ übersetzt werden [Lei99, S.520]. Bilder oder graphische Darstellungen, die einen Gegenstand oder Sachverhalt erklären sollen, werden nach Reichenberger und Steinmetz als *Visualisierungen* bezeichnet. Werden relationale Daten ohne Bezug zu physikalischen Gegenständen, wie z.B. Häusern, Autos etc., visualisiert, so wird von *abstrakten Visualisierungen* oder auch von *Diagrammen* gesprochen [RS99].

Da in dieser Arbeit nur die abstrakte Visualisierung behandelt wird, werden – zur Vereinfachung – die Begriffe *Darstellung*, *Visualisierung*, *abstrakte Visualisierung* und *Diagramm* synonym verwendet.

Es lassen sich auch Definitionen finden, die stärker auf eine Rechnerunterstützung abzielen. In den Frequently Asked Questions (FAQ) zur Internet Newsgroup *comp.graphics.visualization* heißt es: „*Visualization is the use of computer-generated media based on data in the service of human insight/learning.*“ oder ähnlich „*Visualization: The use of computer imagery to gain insign into complex phenomena.*“ [VFAQ98]. Bei der Visualisierung steht derselben FAQ nach der Zweck im Vordergrund, nicht die Ästhetik der Darstellung: „*The purpose of visualization is insight, not virtual realities or pictures*“. Der Visualisierung wird dabei das Ziel zugeschrieben, „*... vorgegebene Daten möglichst mühelos wahrnehmbar, verständlich und überschaubar darzustellen*“ [Eng95, S. 15]. Genügt sie diesem Anspruch nicht, wird sie als „nicht expressiv“ bezeichnet. Hierzu geben Reichenberger und Steinmetz in [RS99] das Beispiel einer Visualisierung, die falsche Interpretationen begünstigt. In Abbildung 3.1 werden Teildisziplinen des Gebietes „Applied Arts” gezeigt. Durch die Anordnung kann es auf dem ersten Blick so erscheinen, als wäre „Furniture Design“ ein Oberbegriff von „Interior Design“. Erst bei näherer Betrachtung wird deutlich, daß statt dessen „Interior Design“ in Kombination mit „Product Design“ zum „Furniture Design“ führt.

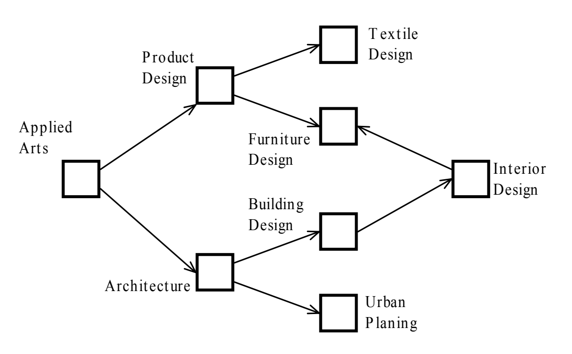


Abbildung 3.1: Beispiel für mangelnde Expressivität (abgewandelt nach [RS99])

Die Expressivität einer Darstellung laßt sich allerdings nicht objektiv quantifizieren und unterliegt letztlich auch dem Empfinden des jeweiligen Betrachters.

3.1.1 Forschungsbereiche

Obwohl, wie erwähnt, bereits seit langem Sachverhalte graphisch dargestellt werden, bildete sich erst in letzter Zeit ein eigenes Forschungsgebiet *Visualisierung* heraus. Häufig findet man eine Einteilung in drei Teilbereiche:

Softwarevisualisierung (*Software Visualization*). Price, Baecker und Small definieren Softwarevisualisierung als „*the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software*" und liefern eine Taxonomie für diese Bereich [PBS93]. Anfänglich wurde in diesem besonders nach Wegen zur Veranschaulichung von Algorithmen gesucht, so daß zunächst vor allem von *Algorithm Visualization* oder auch *Algorithm Animation* gesprochen wurde. Bekannte Systeme hierzu sind der *Brown University Algorithm Simulator and Animator* (BALSA) und Balsa-II [BS84 und Bro88]. Daneben entstand der Begriff *Program Visualization*, der als das Visualisieren von Datenstrukturen sowie Quelltexten verstanden werden kann. Auch die visuelle Programmierung (*visual programming*), d.h. die Spezifikation eines Programmes durch Graphiken, wird zur *Program Visualization* gezählt. Unter Softwarevisualisierung werden diese Termini zusammengefaßt, wobei gleichzeitig eine Lösung vom einzelnen Programm hin zu komplexen Softwaresystemen erfolgt.

Informationsvisualisierung (*Information Visualization*). Eine Charakterisierung liefert Robertson mit „*Information Visualization attempts to display structural relationships and context that would be more difficult to detect by individual retrieval requests.*" [RCM93]. Eine weitere gute Beschreibung liefert Zielonka mit „*Information Visualization uses 3D computer graphics and interactive animation to stimulate recognition of patterns and structure in information.*" [Zie95], allein die Fixierung auf dreidimensionale Graphik erscheint unnötig einschränkend. Da Software eine Vielzahl struktureller Beziehungen enthalten kann, die es darzustellen gilt, sind Erkenntnisse der Informationsvisualisierung für die vorliegende Arbeit hilfreich.

Wissenschaftliche Visualisierung (*Scientific Visualization*). Hier steht die Visualisierung quantitativer Größen, wie sie z.B. bei der Auswertung physikalischer Experimente auftreten, im Vordergrund. Einen Überblick über diese Disziplin liefert z.B. [Bro92].

Da die rechnergestützte Visualisierung das Problem der Kommunikation von Informationen vom Computer zum Menschen umfaßt, findet man Arbeiten zur Visualisierung zudem unter dem Stichwort ***Human Computer Interaction*** (HCI).

3.1.2 Semiotik

Die Semiotik ist die „*Lehre von den sprachlichen Zeichen und ihrer Nachrichtenfunktion*" [Lei99]. In Rahmen dieser Arbeit findet sie bei der Gestaltung der Form der verwendeten Symbole (*Zeichen*) Anwendung. Wesentliche Qualitätsmerkmale für die Verständlichkeit von Zeichen sind deren Unverwechselbarkeit, Originalität, Eindeutigkeit und die Einfachheit ihrer graphischen Form [RS99].

Während es unmittelbar einsichtig ist, warum Unverwechselbarkeit, Originalität und Eindeutigkeit zur Qualität eines Zeichens beitragen, bedarf der Punkt der Einfachheit weiterer Klärung, zumal einleitend die Aussicht, bei dreidimensionaler Computergraphik komplexere und somit intuitivere Symbole verwenden zu können, als ein möglicher Vorteil genannt wurde. Warum ist beispielsweise das häufig in Landkarten verwendete und in Abbildung 3.2a auf nachfolgender Seite gezeigte Symbol für Kirchen einem naturalistischerem und detaillierterem Symbol wie in Abbildung 3.2b vorzuziehen? Ein Grund dafür nennen Reichenberg und Steinmetz in der bereits erwähnten Arbeit [RS99]. Durch eine naturalistische Gestaltung kann die Menge der unerwünscht mit-transportierten Informationen steigen. Beispielsweise könnte das in Abbildung 3.2b gezeigte Symbol so (fehl-)interpretiert werden, daß die Kirchen, die es symbolisiert genau einen Turm haben und in dem gezeigten Baustil erbaut wurden. Das abstraktere Symbol aus 3.2a kann nicht zu solchen Assoziationen führen. Weiterhin können detaillierte Symbole, insbesondere wenn sie in großen Mengen verwendet

werden, dazu führen, daß die Darstellung überfrachtet erscheint und ein Betrachter von den vielen Details „erschlagen“ wird. Somit können zu komplexe Symbole der Verständlichkeit eher abträglich sein. Es gilt also ein Mittelweg zwischen Intuitivität und Komplexität zu finden.

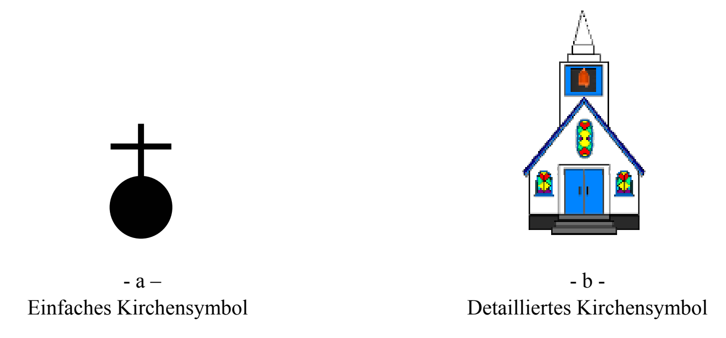


Abbildung 3.2: Zwei verschieden detaillierte Symbole

Der Entwurf dreidimensionaler Zeichen wird dadurch erschwert, daß sie von verschiedenen Betrachtungswinkeln annähernd gleich verständlich sein müssen [PFW98]. Weiterhin ist es im Hinblick auf die veränderliche Betrachtungsposition notwendig, die Zeichen so zu gestalten, daß sie auch noch bei einer gewissen Entfernung erkennbar bleiben.

3.2 Weitere Begriffe

In dieser Arbeit geht es um Visualisierungen, welche die Struktur einer Software zeigen sollen. Diesbezüglich wird nachfolgend auch von ***Softwarestrukturvisualisierungen*** gesprochen.

Unter ***Visualisierungstechniken*** werden im weiteren Text Techniken subsumiert, die Vorgaben für die Gestaltung einer Visualisierung machen. Ein einfaches Beispiel für eine Visualisierungstechnik lieferte bereits Abbildung 3.1 (s. Seite 12) anhand der baumförmigen Darstellung, wenn man den Knoten "Interior Design" und dessen Kanten nicht hinzuzählt.

Unter einem ***Visualisierungssystem*** wird eine Software verstanden, die eine oder mehrere Visualisierungstechniken implementiert, um den Anwender dabei zu unterstützen, Visualisierungen zu erstellen oder zu rezipieren. In diesem Sinne gelten Klassenbrowser als Visualisierungssysteme.

3.3 Dreidimensionale Computergraphik

Wie einleitend erwähnt, wird die dreidimensionale Computergraphik derzeit zunehmend populärer, u.a. aufgrund gesteigerter Hardwareleistungsfähigkeit. War es vor einigen Jahren noch ausschließlich möglich, dreidimensionale Darstellungen aus Linienzügen zu bilden, die an Drahtmodelle erinnerten, so ist heutzutage die Darstellung komplexer Szenen mit simulierter Beleuchtung und dergleichen machbar. Trotzdem stellen Algorithmen zur dreidimensionale Graphik immer noch hohe Anforderungen an die verfügbare Leistung. Derartige Algorithmen entstammen dem Bereich der graphischen Datenverarbeitung zu dem im großen Umfang Literatur publiziert ist. Für Einführungen vgl. z.B. [ZK95] oder [FDF+90]. Nachfolgend werden einige Aspekte aus diesem Zusammenhang eingeführt, soweit sie für diese Arbeit relevant sind.

Graphische Objekte. Dreidimensionale Graphiken – sie werden auch als Szenen bezeichnet – setzen sich aus vielen verschiedenartigen graphischen Objekten zusammen. Um diese darzustellen, werden sie meist in einzelne Dreiecke zerlegt, die leichter von entsprechender Hard- und Software zu verarbeiten sind. Für die Performanz der Darstellung ist es

entscheidend, die Anzahl der benutzten Dreiecke gering zu halten. Rundungen oder auch detaillierte Objekte, wie beispielsweise Texte, lassen sich nur durch die Verwendung vieler Dreiecke annähern und beeinflussen somit die Performanz negativ.

Koordinatensystem. In dieser Arbeit wird mit einem rechtshändigem Koordinatensystem gearbeitet (vgl. Abbildung 3.3). Häufig ist von Ebenen die Rede, die mit zwei der insgesamt drei Koordinatenachsen betitelt werden, z.B. der XY-Ebene. Diese bestehen aus allen Punkten des Raumes, bei denen die Koordinate der nicht genannten Achse – im Beispiel die Z-Achse – gleich null sind.

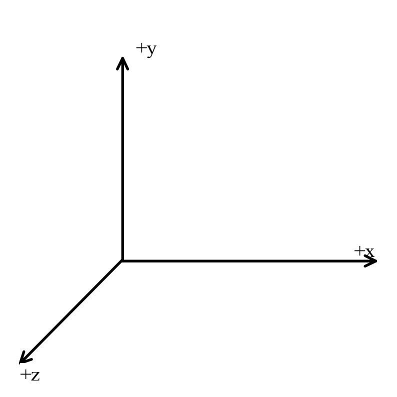


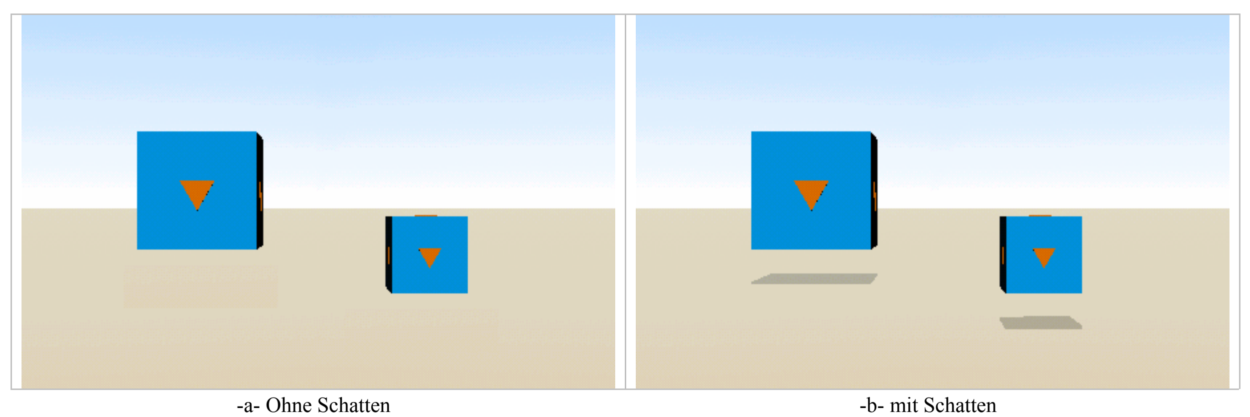
Abbildung 3.3: Verwendetes Koordinatensystem

Virtuelle Kamera. Als Betrachter einer dreidimensionalen Szene möchte man die Ansicht variieren, d.h. z.B. die Szene aus verschiedenen Richtungen betrachten. Software, die diese Möglichkeit bietet, bedient sich häufig der Metapher einer virtuellen Kamera. Die Bildschirmdarstellung wird durch das Bild bestimmt, das die Kamera von der Szene „aufnimmt". Die Darstellungsparameter der Kamera, welche ihre Position im Raum und ihren Blickwinkel umfassen, können vom Betrachter verändert werden. Für diesen entsteht der Eindruck, daß er sich mit der virtuellen Kamera quasi selbst durch die Szene bewegt. In diesem Sinne werden in dieser Arbeit Begriffe wie Betrachtungsposition und dergleichen verwendet.

Hilfen für die Anwendungsentwicklung. Das Erstellen von Software für dreidimensionale Computergraphik ist vergleichsweise aufwendig. Deshalb sind Hilfen entstanden, die den Programmierer bei dieser Aufgabe unterstützen. Als Beispiele können hier die *Virtual Reality Modeling Language* (VRML) und die *Java 3D API* dienen. Bei VRML werden textuell vorliegende Dateien durch einen sog. VRML-Browser interpretiert und angezeigt (vgl. z.B. [ANM97]). Dieser Browser kann innerhalb der Anwendung als eine Komponente zur Darstellung dreidimensionaler Graphiken genutzt werden. Die Java 3D API dagegen ist eine Erweiterung der Klassenbibliothek von Java, die von Sun kostenlos im Internet zur Verfügung gestellt wird [J3D]. Auf VRML wird, da die Sprache für diese Arbeit verwendet wurde, im Rahmen des dritten Teils dieser Arbeit näher eingegangen.

Unterstützung des Tiefeneindrucks. Ein besonderer Vorteil dreidimensionaler Darstellungen ist es, daß bei ihnen die „Tiefe" des Darstellungsraumes genutzt werden kann, während bei zweidimensionalen Darstellungen nur eine Ebene zur Verfügung steht. Dabei ist es entscheidend, dem Betrachter einen Eindruck von der Tiefe zu verschaffen, in der ein bestimmter Bestandteil einer Darstellung plaziert ist. Dieser kann bei der Projektion des dreidimensionalen Diagramms in eine Ebene, wie z.B. einen Bildschirm, teilweise verloren gehen. Deshalb sind verschiedene Techniken entwickelt worden, welche das Vermitteln eines Tiefeneindrucks unterstützen sollen. Eine umfassende Betrachtung dieser Unterstützungstechniken findet sich beispielsweise in [XM92]. Die Techniken basieren teilweise auf spezieller Hardware, wie z.B. Head-Mounted-Displays und Eye-Shutter-Brillen. Es existieren aber auch Methoden, die ohne Hardwareunterstützung verwendet werden können. Hier sind z.B. eine „Beleuchtung" der Szene aus simulierten Lichtquellen oder ein simulierter Nebel, der quasi „über die Darstellung gelegt wird" zu nennen. Ein gute Unterstützung des Tiefeneindrucks ergibt sich auch durch eine permanente Bewegung der Darstellung. Die Eignung einer permanenten Drehung wurde in einer Studie erwiesen [WF96]. Weiterhin hilft eine Schattierung der Szene. Schatten, die durch Objekte auf andere Objekte geworfen werden, erfordern viel Rechenzeit und werden zudem durch gängige Hilfen für die Anwendungsentwicklung nicht unterstützt. Es hat sich aber gezeigt, daß insbesondere mittels der Verwendung von Schatten auf einem „Boden" der Tiefeneindruck gut unterstützt werden

kann (vgl. Abbildung 3.4). Deren Simulation ist vergleichsweise unaufwendig.



-a- Ohne Schatten -b- mit Schatten

Abbildung 3.4: Unterstützung des Tiefeneindrucks durch Schatten

Nachdem nun die Motivation für diese Arbeit beschrieben wurde und einige Grundlagen eingeführt wurden, kann im nachfolgenden Kapitel als erster Schritt der Konzeption das Ziel der Arbeit präzisiert werden.

Teil II

Konzeption

4 Ziel und Vorgehen

4.1 Ziel

Ziel dieser Arbeit ist es, ein Konzept für einen Klassenbrowser für Java-Software aufzuzeigen, der das Anfertigen und Betrachten von dreidimensionalen und expressiven Softwarestrukturvisualisierungen ermöglicht. Zudem soll die Realisierbarkeit dieses Konzepts gezeigt werden.

Da Visualisierungen vor allem bei komplexen Sachverhalten sinnvoll sind, muß der Klassenbrowser auch bei Software einsetzbar sein, deren Umfang über triviale Beispiele hinausgeht. Daraus ergeben sich zwei grundlegende Notwendigkeiten:

a) *Es müssen Wege gefunden werden, die Darstellung sinnvoll zu strukturieren und ggf. zu filtern, um einer Überfrachtung entgegenzuwirken.*

   Dabei gilt es, erwartete Vorteile der dreidimensionalen Visualisierung wie Natürlichkeit etc. (vgl. Kapitel 2.4) nach Möglichkeit auszunützen, um expressive Visualisierungen zu erhalten. Wie im Abschnitt 3.1 erwähnt, ist die Expressivität einer Visualisierung jedoch zu einem gewissen Teil subjektiv. Maßnahmen, welche die Expressivität in vielen Fällen begünstigen, sollten sich dennoch finden lassen.

b) *Es müssen Mittel bereitgestellt werden, die den Aufwand für das Anfertigen von Visualisierungen in einem vertretbaren Rahmen halten.*

   Welcher Aufwand noch „vertretbar" ist, läßt sich allerdings nicht allgemeingültig festlegen. Abhängig von der Größe der betrachteten Software und auch des Zwecks der Visualisierung – soll während der Entwicklung einer Software „mal eben" der aktuelle Stand gezeigt werden oder dient die Visualisierung der Schulung von Kunden – sind hier unterschiedliche Maßstäbe anzusetzen. Ein zügiges Arbeiten sollte aber in allen Fällen unterstützt werden. Beispielsweise ist ein Vorgehen, bei dem mittels eines herkömmlichen dreidimensionalen graphischen Zeichenprogramms eine Visualisierung ausschließlich von Hand erstellt wird, so daß für jede Klasse ein Symbol plaziert und für jede Beziehung ein Pfeil gezeichnet werden müßte, aufgrund der zu erwartenden Vielzahl von Klassen und Beziehungen nicht praktikabel.

Das Konzept für den Klassenbrowser besteht aus Antworten auf die folgenden drei Fragen:

1) Welche Sprachkonzepte von Java sollen visualisiert werden?
2) Wie sollen Softwarestrukturvisualisierungen aufgebaut sein?
3) Wie ist die Benutzung des Klassenbrowser zu gestaltet, um den Anfertigungsaufwand für Visualisierungen zu verringern?

In diesem zweiten Teil der Arbeit wird schwerpunktmäßig auf die ersten beiden Fragen eingegangen. Die Antwort auf die erste Frage findet sich in der Definition eines Metamodells für sogenannte *Strukturmodelle* von Java-Software. Das Strukturmodell einer Software bündelt Informationen über ihren Aufbau und dient somit als Grundlage für ihre Visualisierung. Das Metamodell für Strukturmodelle wird im nachfolgenden Kapitel 5 definiert. Da es festlegt, welche Informationen über Java-Software berücksichtigt werden, kann es als Anforderungskatalog an den Teil der Konzeption angesehen werden, der sich mit dem Aufbau von Visualisierungen beschäftigt.

Um zu entscheiden, wie Visualisierungen aufgebaut sein sollen, ist es naturgemäß hilfreich, die einschlägige Literatur zu studieren. Dabei stellt sich heraus, daß bereits eine Reihe von Visualisierungstechniken existieren. Aus dieser Erkenntnis entstand die grundsätzliche Idee

zur Beantwortung der zweiten Frage: Zur Visualisierung sollen nach Möglichkeit vorhandene Visualisierungstechniken genutzt werden. Daher umfaßt die Konzeption des Klassenbrowsers die Vorstellung derartiger Techniken (vgl. Kapitel 7). Weiterhin gilt es, Techniken für den Einsatz zur Darstellung von Softwarestrukturen auszuwählen und ggf. Rahmenbedingungen aufzuzeigen, welche die Zweckmäßigkeit eines Einsatzes beeinflussen (vgl. Kapitel 8). Dabei wird sich herausstellen, daß Abänderungen und Ergänzungen zu den Techniken aus der Literatur notwendig sind. Diese werden gleichermaßen im Kapitel 8 erläutert. Dort wird sich ebenfalls zeigen, daß es sinnvoll ist, verschiedene Techniken zu kombinieren.

Im dritten Teil dieser Arbeit wird durch eine prototypische Implementierung die Realisierbarkeit des Konzeptes gezeigt. Dabei wird anhand der Vorstellung einer Benutzungsoberfläche für den Klassenbrowsers auch eine Anwort auf die letzte konzeptionelle Frage gegeben.

4.2 Vorgehen

Um das genannte Ziel zu erreichen, wurden vier Phasen durchlaufen (vgl. Übersicht in Abbildung 4.1). In der ersten Phase – der *Vorbereitung* – galt es zunächst festzulegen, welche Sprachkonzepte von Java visualisiert werden sollen. Diese Festlegung führte zur Definition des Metamodells für Strukturmodelle. Anschließend wurden dann Beispiele für Java-Software gesucht, anhand derer ein Konzept zur Visualisierung entwickelt werden konnte. Diese Beispiele werden im Kapitel 6 nach der Vorstellung der Metamodells charakterisiert, da hierzu einige der mit dem Metamodell eingeführten Begriffe notwendig sind.

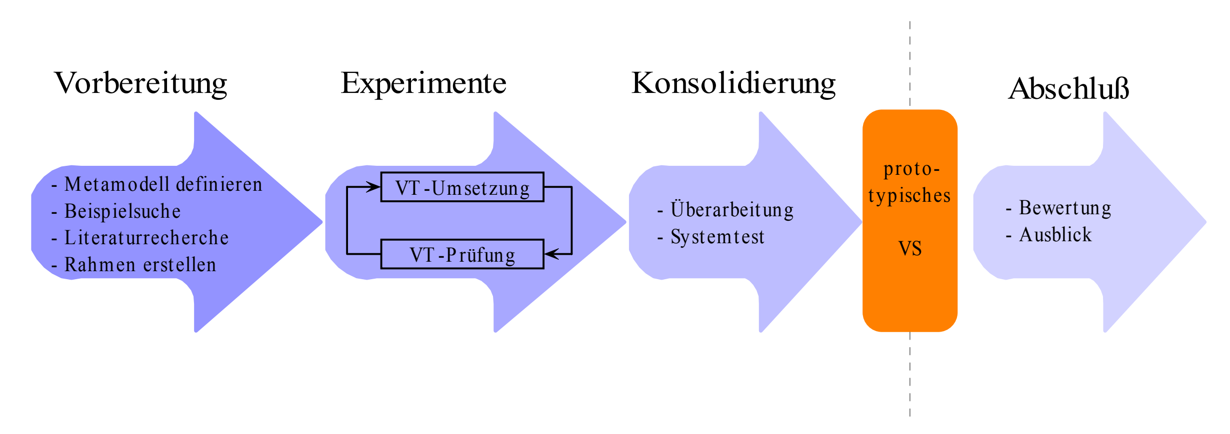


Abbildung 4.1: Übersicht über das Vorgehen

Als dritten Schritt während der Vorbereitung wurde eine Literaturrecherche betrieben, bei der es vor allem darum ging, vorhandene Ansätze zur Visualisierung aufzuarbeiten. Hierbei wurde die bereits erwähnte Idee gewonnen, verschiedene Techniken kombiniert einzusetzen. Es galt somit zu ermitteln, welche Technik unter welchen Umständen besonders geeignet ist. Dies war Aufgabe der zweite Phase – der *Experimentierphase*. Die Eignungsprüfungen konnten letztlich nur erfolgen, indem eine Auswahl der Techniken, für die eine besondere Aussicht auf Brauchbarkeit bestand, probeweise implementiert wurde. Um dies zu erleichtern, ist als letzter Schritt der Vorbereitung ein einfacher Rahmen geschaffen worden, der stets wiederkehrende Aufgaben, wie z.B. das Anzeigen einer dreidimensionalen Szene, übernimmt. In den Rahmen konnten die Implementierungen der verschiedenen Techniken eingebettet werden.

In der Experimentierphase selbst wurde zwischen zwei Tätigkeiten iteriert. Zunächst wurde die Zweckmäßigkeit von Visualisierungstechniken geprüft. Dabei ergaben sich einige Ansätze für Verbesserungen sowie Ideen für weitere Techniken, nach deren Umsetzung ein erneuter Beurteilungsbedarf bestand. Die Iteration wurde durchgeführt, bis ein Stand erreicht war, der eine expressive Visualisierung zumindest größerer Ausschnitte der Beispiele ermöglichte – allerdings bei subjektiver Bewertung.

In der sich anschließenden *Konsolidierungsphase* wurde der vorhandene Rahmen zusammen mit den eingebetteten Visualisierungstechniken überarbeitet. Dabei wurde das Design in einigen Punkten redigiert sowie eine verbesserte Benutzungsschnittstelle erstellt, so daß aus dem Rahmen ein – noch immer prototypisches – Visualisierungssystem entstand. Das Testen dieses Systems war ebenfalls Bestandteil der Konsolidierungsphase. Zudem hatte sich während der Experimentierphase gezeigt, daß auch am Metamodell einige kleinere Änderungen sinnvoll sind.

Den *Abschluß* dieser Arbeit bildeten eine Bewertung des Erreichten und ein Ausblick auf weitere Entwicklungsmöglichkeiten (vgl. Teil IV).

5 Strukturmodelle für Java-Software

Bevor die Frage untersucht werden kann, wie Visualisierungen von Softwarestrukturen sinnvoll aufzubauen sind, muß zunächst geklärt werden, welche Aspekte in solchen Visualisierungen überhaupt zu berücksichtigen sind. Als Antwort auf die Frage „*Welche Sprachkonzepte von Java sollen visualisiert werden?*“ wird in diesem Kapitel ein Metamodell für sogenannte Strukturmodelle entwickelt. Strukturmodelle beschreiben die Anwendung verschiedener in Java zur Verfügung stehender Konzepte zur Gliederung einer Software. Das Metamodell wiederum legt fest, welche Konzepte in Strukturmodellen berücksichtigt werden.

Für das hier vorgestellte Metamodell wird kein Anspruch auf Vollständigkeit erhoben. Beispielsweise werden dynamische Aspekte von Software, wie der Kontrollfluß durch verschiedene Methoden, bewußt außen vor gelassen, um das Thema dieser Arbeit stärker zu fokussieren. Die Aufgabe eines Klassenbrowser wird in der Vermittlung eines Überblicks über statische Aspekte, wie z.B. der deklarierten Klassenhierarchien oder der vorgenommen Einteilungen in Subsysteme, gesehen. Diese Sicht wird um einige Details angereichert, die für das Verständnis des Sachverhalts als wichtig erachtet werden. Aus Aufwandsgründen wird sich dabei auf die Berücksichtigung von Sprachkonzepten beschränkt, die typischerweise auch von herkömmlichen zweidimensionalen Klassenbrowsern unterstützt werden. An diversen Stellen könnten daher Erweiterungen, die zu einer detaillierteren Modellierung einer Java-Software führen würden, durchaus noch möglich und sinnvoll sein, als Basis für diese Arbeit hat sich der hier gezeigte Umfang des Metamodells aber bewährt.

Für eine Einführungen in die Programmierung mit Java sei auf die umfangreiche Literatur zu diesem Thema verwiesen (vgl. z.B. [Küh96]). Die Darstellung hier orientiert sich an der Sprachdefinition von Java [GJS96].

Abbildung 5.1 zeigt mittels der Unified Modeling Language (UML) [BRJ99, Oes98] einen ersten Überblick über das Metamodell für Strukturmodelle. Ein Strukturmodell modelliert Java-Software. Diese besteht aus der Anwendung von Sprachkonzepten, die Java zur Verfügung stellt. Die Anwendung führt zu *Gegebenheiten* innerhalb der Software. Diese können grob unterteilt werden in *Elemente*, in die eine Software gegliedert wird, und *Beziehungen* zwischen diesen Elementen.

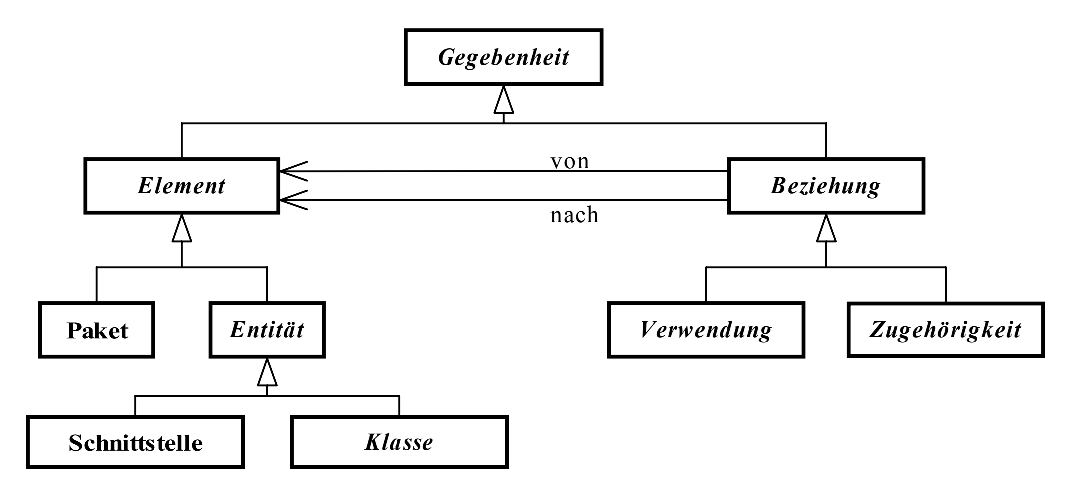


Abbildung 5.1: Überblick über das Metamodell für Strukturmodelle
Kursiv geschriebene Namen kennzeichnen in dieser und in den folgenden Abbildungen abstrakte Klassen gegenüber konkreten Klassen (siehe auch Abschnitt 5.1.3).

Ein zentrales Konzept der objektorientierten Programmierung ist das Bilden von *Klassen*. Wie in vielen anderen objektorientierten Programmiersprachen ist dies auch in Java möglich.

Nicht so weit verbreitet sind *Schnittstellen*, durch die abstrakte Datentypen spezifiziert werden können. Da Klassen und Schnittstellen viele Gemeinsamkeiten aufweisen, werden sie in diesem Text unter dem Begriff der *Entität* zusammengefaßt. Entitäten können nach beliebigen Gesichtspunkten zu *Paketen* gebündelt werden, um Software in Subsysteme aufzuteilen.

Zwischen den Elementen einer Software können verschiedenartige Beziehungen bestehen. Diese lassen sich zunächst in *Zugehörigkeiten* und *Verwendungen* unterteilen. Von einer Zugehörigkeit wird gesprochen, wenn ein Element Teil eines anderen ist. Beispielsweise wird das Bündeln von Entitäten zu einem Paket über Zugehörigkeiten modelliert, d.h. Entitäten gehören zu jeweils einem Paket. Von einer Verwendung wird in diesem Text gesprochen, wenn bei der Deklaration eines Elements auf ein anderes Element verwiesen wird. Beispielsweise können, wie in objektorientierten Programmiersprachen üblich, Klassen voneinander erben. Weitere Beispiele werden nachfolgend noch gegeben.

Im folgenden werden zunächst Klassen näher betrachtet, daran anschließend Schnittstellen (Abschnitte 5.1 und 5.2). Dabei werden auch zwei Beziehungsarten vorgestellt: Erweiterungen und Implementierungen. Über die Deklaration von Klassen und Schnittstellen können Typen geschaffen werden. Auf weitere mögliche Typen wird im Abschnitt 5.3 eingegangen. Im Abschnitt 5.4 werden Pakete sowie Importe zwischen Paketen erläutert. Eine Besonderheit von Java ist die Möglichkeit, Deklarationen von Entitäten ineinander zu schachteln. Dies wird im Abschnitt 5.5 gezeigt. Abschnitt 5.6 geht auf sogenannte Ausnahmen (*Exceptions*) ein. Abschnitt 5.7 klärt, wann von einer Benutzung einer Entität durch eine andere gesprochen wird. Abschließend wird in Abschnitt 5.8 das entwickelte Metamodell noch einmal als Ganzes gezeigt.

5.1 Klassen

Wie z.B. auch bei Smalltalk oder Eiffel besteht Java-Software in der Hauptsache aus Deklarationen von Klassen, die Schablonen für gleichartige Objekte darstellen. Objekte sind Instanzen von Klassen. Innerhalb der Deklaration einer Klasse können u.a. Konstruktoren, Methoden und Variablen[1] deklariert werden.

Nachfolgendes Beispiel 5.1 zeigt die Deklaration einer Klasse *Fahrzeug* zusammen mit der einer Klasse *Test*, deren Methode *t* eine Instanz von *Fahrzeug* erzeugt und eine Methode dieser Instanz aufruft. Wie man am Beispiel sieht, erinnert die Syntax von Java an C++.

Beispiel 5.1: **[Klassendeklaration]**

```
class Fahrzeug {
  String standort; // Variable mit aktuellem Standort als Zeichenkette

  Fahrzeug(String initialeStandort) {      // Konstruktor
    standort = initialerStandort;
  }

  String fahren(String ziel) {             // Methode
    standort = ziel; return standort;
  }
}
```

[1] In der Java-Sprachdefinition wird in diesem Zusammenhang von „Fields" (Feldern) gesprochen. Da unter dem Begriff „Feld" in der Informatik häufiger eine „Aneinanderreihung gleichartiger Elemente" verstanden wird (vgl. [Eng93, S.248]), wird in dieser Arbeit – wie z.B. auch in [Küh96] – von „Variablen" gesprochen. Variablen, die innerhalb eines Codeblock deklariert werden, werden zur Unterscheidung immer als „lokale Variablen" bezeichnet.

```
class Test {
  static void t() {                         // statische Methode
    Fahrzeug f = new Fahrzeug("Dortmund"); f.fahren("Essen")
  }
}
```

Instanzen der Klasse *Fahrzeug* besitzen die Variable *standort* vom Typ *String* und die Methode *fahren*. Neue Instanzen von *Fahrzeug* können unter Verwendung eines Konstruktors erzeugt werden, der als Parameter einen initialen Standort erwartet.

Weiterhin können innerhalb von Klassen deklariert werden:

- *Innere Entitäten* (vgl. Abschnitt 5.5).
- *Statische Methoden und Variablen*. Neben Variablen und Methoden, welche den Instanzen einer Klasse zugeordnet werden, können auch Klassen selbst über Variablen und Methoden verfügen. Diese werden durch das Schlüsselwort *static* gekennzeichnet und können verwendet werden, ohne daß eine Instanz der entsprechenden Klasse vorliegt. Beispielsweise kann der Aufruf der Methode *t* von *Test* aus dem Beispiel 5.1 durch *Test.t*() erfolgen.
- *Unveränderliche Variablen (Konstanten)*. Diese werden durch das Schlüsselwort *final* von veränderlichen Variablen unterschieden. Ein Initialisierungsausdruck, der bei Variablendeklarationen optional ist, ist hier zwingend erforderlich (vgl. Beispiel 5.2). Im Regelfall sind Konstanten gleichzeitig statisch.
- *Instanzinitialisierer*. Diese bestehen aus einem einzelnen Codeblock, der immer dann ausgeführt wird, wenn eine neue Instanz einer Klasse erzeugt wird (vgl. Beispiel 5.2).
- *Klasseninitialisierer*. Diese bestehen ebenfalls aus einem einzelnem Codeblock, dem das Schlüsselwort *static* vorangestellt wird. Sie werden ausgeführt, wenn erstmals nach dem Start eine Software auf eine Klasse zugegriffen wird (vgl. Beispiel 5.2).

Beispiel 5.2: [Kostanten, Klassen- und Instanzinitialisierer]

```
class EineKlasse {
  static final int KONSTANTE = 10;             // Konstante
  static { ... Initialisierung der Klasse ... }// Klasseninitialisierer
  { ... Initialisierung einer Instanz ... }    // Instanzinitialisierer
}
```

Konstruktoren, Methoden, Klassen- und Instanzinitialisierer sowie Variablen werden nachfolgend unter dem Begriff der *Entitätselemente* zusammengefaßt.

5.1.1 Erweiterungen zwischen Klassen

Wie in vielen objektorientierten Programmiersprachen ist die Vererbung in Java ein wichtiges Sprachkonzept. Vererbung wird in Java dadurch erreicht, daß eine Klasse eine andere erweitert. Nachfolgendes Beispiel zeigt die Deklaration einer Klasse *Auto*, welche die oben deklarierte Klasse *Fahrzeug* erweitert:

Beispiel 5.3: **[Erweiterung von Klassen]**

```
class Auto extends Fahrzeug {
  Auto(String initialerStandort) { super(initialerStandort);tanken(); }
  int  tankfuellung;
  void tanken() { tankfuellung = 50; }
}
```

Die Klasse, die erweitert wird, ist nach dem Schlüsselwort *extends* angegeben. Sie wird als die direkte Superklasse der deklarierten Klasse bezeichnet. Umgekehrt ist die deklarierte Klasse eine direkte Subklasse der hinter *extends* angegebenen Klassen.

Eine direkte Subklasse erbt die Methoden und Variablen ihrer direkten Superklasse und kann zusätzlich weitere Entitätselemente deklarieren. Jede Instanz einer erweiternden Klasse kann durch die geerbten Methoden und Variablen als Instanz der erweiterten Klasse angesehen werden.

In Java wird keine Mehrfacherbung unterstützt, d.h. eine Klasse kann maximal eine direkte Superklasse haben. Genauer besitzen Klassen in Java exakt eine direkte Superklasse. Zwar ist die Verwendung von *extends* bei der Deklaration einer Klasse optional, wird aber keine direkte Superklasse angegeben, so ist die besondere Klasse *Object* die direkte Superklasse. Diese entstammt der zu Java gehörenden Klassenbibliothek. Sie ist die einzige Klasse in Java, die keine direkte Superklasse besitzt.

Die Erweiterung ist transitiv, d.h. Klassen sind nicht nur Erweiterungen ihrer jeweiligen direkten Superklasse, sondern auch von deren direkter Superklasse usw. Dies drückt sich in der Definition der Begriffe *Subklasse* und *Superklasse* aus. Eine Klasse *A* ist genau dann Subklasse von *C*, wenn

- *A* direkte Subklasse von *C* ist, oder
- eine Klasse *B* existiert, mit *A* Subklasse von *B* und *B* Subklasse von *C*.

Weiterhin heißt *C* genau dann Superklasse von *A*, wenn *A* Subklasse von *C* ist.

Klassen bilden zusammen mit ihren Erweiterungsbeziehungen eine Hierarchie. Dementsprechend wird nachfolgend auch von Klassen-, Erweiterungs- oder auch Vererbungshierarchien gesprochen.

5.1.2 Redefinitionen

Eine Erweiterung kann mit Redefinitionen von Methoden einher gehen. Eine Redefinition liegt vor, wenn in einer Subklasse eine nicht-statische Methode deklariert wird, welche denselben Namen und dieselben formalen Parameter besitzt, wie eine bereits in einer Superklasse vorhandene nicht-statische Methode. Nachfolgendes Beispiel zeigt eine modifizierte Deklaration der Klasse *Auto*.

Beispiel 5.4: [Redefinition]

```
class Auto extends Fahrzeug {
  Auto(String initialerStandort) { super(initialerStandort);tanken(); }
  int    tankfuellung;
  void   tanken() { tankfuellung = 50; }
  String fahren(String ziel) { tanken(); return super.fahren(ziel); }
}
class Fahrtenplaner {
  void fahrtAusführen(Fahrzeug f,String ziel) { f.fahren(ziel); }
}
```

Hier wird die Methode *fahren* redefiniert. Deren verändertes Verhalten besteht darin, daß zunächst die Methode *tanken* aufgerufen wird, bevor anschließend über das Konstrukt *super.fahren(ziel)* die ursprüngliche Implementierung aufgerufen wird. Redefinition ist in Java ein mächtiges Instrument – wie bei der objektorientierten Programmierung insgesamt – da sich hierdurch ein Objekt, welches eine Methode aufruft, nicht um die konkrete Klasse des

Objektes kümmern muß, dessen Methode aufgerufen wird. Dies wird in der Klasse *Fahrtenplaner* des Beispiel 5.4 gezeigt. Da überall dort, wo eine Instanz von *Fahrzeug* erwartet wird, auch eine Instanz von *Auto* angegeben werden kann, kann einem Aufruf von *fahrtAusführen* auch eine Instanz von *Auto* als Parameter *f* zugewiesen werden. Wenn der Aufruf von *fahren* für *f* erfolgt, entscheidet das Laufzeitsystem von Java, welche Implementierung von *fahren* verwendet wird, d.h. ob vor der Fahrt getankt wird oder nicht.

5.1.3 Abstrakte, konkrete und finale Klassen

In Java kann das Vorhandensein einer Methode in einer Klasse gefordert werden, ohne daß die Methode durch einen Codeblock implementiert wird. Es wird dann von einer abstrakten Methode gesprochen. Klassen, die mindestens eine abstrakte Methode deklarieren, werden als *abstrakte Klassen* bezeichnet. Derartige Klassen können nicht instanziiert werden. Abstrakte Methoden und Klassen werden durch das Schlüsselwort *abstract* gekennzeichnet. In eine Subklasse einer abstrakten Klasse müssen die abstrakten Methoden unter Angabe eines Codeblock redefiniert werden, ansonsten ist die Subklasse ebenfalls abstrakt. Bei der Redefinition von abstrakten Methoden wird auch von deren Implementierung gesprochen. Klassen, die nicht abstrakt sind, werden als *konkrete Klassen* bezeichnet. Folgendes Beispiel zeigt die Deklarationen der abstrakten Klasse *Reservierung* und deren konkreter Subklasse *TagesReservierung*.

Beispiel 5.5: [Abstrakte und konkrete Klassen]

```
abstract class Reservierung {
  abstract void durchführen(Fahrzeug f);
}
class TagesReservierung extends Reservierung {
  void druchführen(Fahrzeug f) { ... }
}
```

Zusätzlich kann in Java verhindert werden, daß zu einer bestimmten Klasse Subklassen erzeugt werden. Dies ist nur bei konkreten Klassen zulässig. Es geschieht über die Angabe des Schlüsselwortes *final* vor *class*, so daß in diesem Fall von *finalen Klassen* gesprochen wird. Auch einzelne Methoden können als final gekennzeichnet werden, um deren Redefinition im Einzelnen zu verhindern.

5.1.4 Zugriffsmodi

Im Sinne eines Information Hiding möchte man nicht alle Entitätselemente einer Klasse nach außen sichtbar machen. Dazu werden in Java die Zugriffsmodifizierer *public*, *private* und *protected* verwendet, die bei der Deklaration eines Entitätselements angegeben werden können und dessen Zugriffsmodus bestimmen. Folgende Tabelle 5.1 zeigt die Semantik der verschiedenen Zugriffsmodi, wobei der Modus *default* vorliegt, wenn kein Zugriffsmodifizierer angegeben wurde.

Zugriff	Semantik
private	Entitätselement ist nur innerhalb der Entität sichtbar.
default	Entitätselement ist innerhalb des Paketes der Entität sichtbar.
protected	Entitätselement ist innerhalb des Paketes der Entität und innerhalb aller erweiternden Entitäten sichtbar.
public	Entitätselement ist uneingeschränkt nach außen sichtbar.

Tabelle 5.1: Semantik der Zugriffmodi

Man erkennt, wie die Beschränkungen ausgehend von *private* immer weiter gelockert werden. In diesem Sinne wird in vorliegender Arbeit von einer Rangordnung der Zugriffsmodi gesprochen. Für diese gilt:

$$private < default < protected < public.$$

Auch für Klassen selbst kann ein Zugriffsmodus bestimmt werden. Normalerweise sind hier nur die Modi *public* und *default* möglich. Eine Ausnahme bilden innere Entitäten, auf die im Abschnitt 5.5 eingegangen wird. Nachfolgendes Beispiel deklariert eine uneingeschränkt sichtbare Klasse *Autovermietung* mit der nach außen sichtbaren Methode *reservieren* und der nur intern sichtbaren Methode *datenSpeichern*.

Beispiel 5.6: [Sichtbarkeit]

```
public class Autovermietung {
  public  void reservieren() { ... }
  private void datenSpeichern() { ... }
}
```

5.1.5 Ausführbarkeit

Java-Software kann in mehreren Formen auftreten. Bekannte Formen sind eigenständige Applikationen sowie Applets. Weitere Formen sind z.B. Doclets für das zum Java Development Kit gehörende Dokumentierungswerkzeug Javadoc [JDK] oder Servlets [HC98]. Als Einstiegspunkt in eine Applikationen dient eine Klasse, die eine Methode *main* deklariert. Die Ausführung eines Applets beginnt bei einer Klasse, die Subklasse der Klasse *Applet* ist. Klassen, die derartig zum Start einer Software genutzt werden können, werden nachfolgend als *ausführbar* bezeichnet.

5.1.6 Modellierung von Klassen und Erweiterungen

Abbildung 5.2 zeigt, wie Klassen und Erweiterungen in das Metamodell integriert werden. Jede Klasse verfügt über einen Namen. Da alle Elemente benannt sind, besitzt bereits *Element* ein Attribut *name*. Für jede Klasse wird weiterhin ihre Ausführbarkeit, ihre Instanziierbarkeit und ihr Zugriffsmodus gespeichert. Die letzten beiden Eigenschaften sind – wie noch gezeigt wird – auch für Schnittstellen relevant, so daß entsprechende Assoziationen bereits bei *Entität* vorgesehen sind. Mit *Instanziierbarkeit* und *Zugriff* werden zwei Aufzählungstypen modelliert. Entsprechend gilt, daß von den drei bzw. vier Attributen stets genau eins mit *true* und alle anderen mit *false* belegt sind. Warum *Klasse* abstrakt ist, wird ersichtlich, wenn im Abschnitt 5.6 auf sogenannte Ausnahmen (*Exceptions*) eingegangen wird.

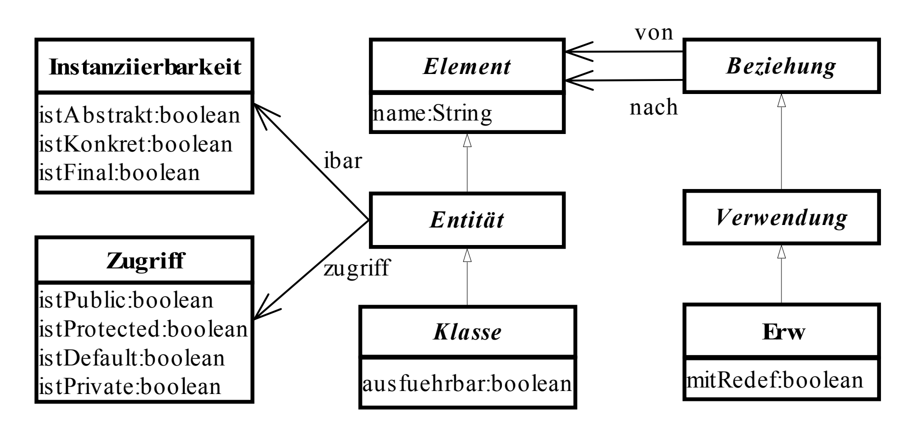


Abbildung 5.2: Modellierung von Klassen und Erweiterungen

Bei Erweiterungen handelt es sich um eine Form der Verwendung. Erweiterungen werden durch *Erw* modelliert, wobei für eine Instanz *erw* von *Erw* gilt: *erw.von* verweist auf die di-

rekte Subklasse und *erw.nach* auf die direkte Superklasse. Dabei wird festgehalten, ob Redefinitionen vorliegen. Genauer gesagt, wird auf die Modellierung der direkten Sub- bzw. Superklasse verwiesen, nicht auf die Klasse selbst. Zur Vereinfachung wird dieser Umstand nachfolgend nicht mehr hervorgehoben.

5.2 Schnittstellen

Neben Klassen bietet Java auch sogenannte Schnittstellen. Diese dienen der Spezifikation von abstrakten Datentypen. Bei abstrakten Datentypen werden Wertemengen ausschließlich durch die darauf zulässigen Operationen beschrieben, ohne daß auf die Implementierung der Operationen eingegangen wird (vgl. z.B. [Eng93, S.173ff]). Schnittstellen bestehen daher aus Deklarationen abstrakter Methoden. Zusätzlich können unveränderliche Variablen (Konstanten) definiert werden. Da Schnittstellen und ihre Methoden immer abstrakt und ihre Variablen immer unveränderlich sind, soll die Angabe der Schlüsselwörter *abstract*, *final* und *static* gemäß Sprachdefinition unterlassen werden. Methoden und Konstanten von Schnittstellen haben immer den Zugriffsmodus *public*, so daß auch der Zugriffsmodifizierer entfallen kann. Schnittstellen selbst können aber verschiedene Zugriffsmodi haben.

Beispiel 5.7: [Deklaration einer Schnittstelle mit Zugriffsmodus default]

```
interface Ausdruckbar {
  int MAXIMALE_SEITENZAHL = 10;   // Konstante         (public)
  void drucken();                 // abstrakte Methode (public)
}
```

Es ist zu beachten, daß in Schnittstellendeklarationen nicht alle Arten von Entitätselementen möglich sind. Konstruktoren, Klassen- und Instanzinitialisierer sowie nicht-statische Variablen sind nicht möglich.

In Strukturmodellen werden Schnittstellen über *Schnittstelle* modelliert; einer Spezialisierung von *Entität*, die über keine zusätzlichen Attribute oder Assoziationen verfügt.

5.2.1 Erweiterungen zwischen Schnittstellen

Auch bei Schnittstellen kann eine Erweiterung stattfinden. Wiederum wird diese durch das Schlüsselwort *extends* angezeigt, und es wird von direkten Super- bzw. Subschnittstellen gesprochen. Die Verallgemeinerung von den Begriffen direkte Super- und direkter Subschnittstelle zu Super- und Subschnittstelle erfolgt analog der bei Klassen gezeigten Definition. Allerdings kann eine Schnittstelle mehrere Schnittstellen erweitern. Auch bei der Erweiterung zwischen zwei Schnittstellen kann es zu Redefinitionen kommen.

Die Modellierung von Erweiterungen zwischen Schnittstellen erfolgt wie bei der von Erweiterungen zwischen Klassen über *Erw* (vgl. Abbildung 5.2 auf vorheriger Seite).

Analog zu Klassen wird gleichfalls bei Schnittstellen im folgenden von Erweiterungs-, Vererbungs- oder auch Schnittstellenhierarchien gesprochen.

5.2.2 Implementierung von Schnittstellen

Eine Klasse kann mehrere Schnittstellen implementieren. Dies wird über das Schlüsselwort *implements* bei der Klassendeklaration angegeben. Die bei der Deklaration einer Klasse angegebenen Schnittstellen werden als direkte Superschnittstellen der Klasse bezeichnet. Wenn eine Schnittstelle *X* direkte Superschnittstelle einer Klasse *A* ist und *Y* Superschnittstelle von *X*, dann ist *Y* Superschnittstelle von *A*. Um eine Schnittstelle zu implementieren, müssen ihre Methoden durch die jeweilige Klasse implementiert werden. Geschieht dies nicht vollständig, so ist die Klasse abstrakt. Implementiert eine Klasse eine Schnittstelle, dann implementierte sie auch alle deren Superschnittstellen.

Im nachfolgenden Beispiel wird eine Schnittstelle *Ausdruckbar* deklariert, die durch die Klasse *Rechnung* implementiert wird. Die Deklaration der Klasse *Druckerwarteschlange* zeigt die Verwendung einer Schnittstelle.

Beispiel 5.8: [Implementierung und Verwendung von Schnittstellen]

```
class Rechnung implements Ausdruckbar { void drucken() {...} }
class Druckerwarteschlange { void einfuegen(Ausdruckbar a) {...} }
```

Implementierungen von Schnittstellen werden im Metamodell durch eine weitere Spezialisierung von *Verwendung* mit Namen *Impl* modelliert, die keine zusätzlichen Attribute oder Assoziationen besitzt. Für eine Instanz *impl* von *Impl* gilt: *impl.von* verweist auf die direkte Subklasse und *impl.nach* auf die direkte Superschnittstelle.

5.3 Typen

In vorangegangenen Abschnitten wurde bereits mehrfach von Typen gesprochen. In Java ist jedem Wert ein Typ zugewiesen. Dabei wird zwischen primitiven Typen und Referenztypen unterschieden. Ein Beispiel für einen primitiven Typ ist *int*, der für ganze Zahlen vorgesehen ist. Die Menge der primitiven Typen ist im Sprachumfang von Java definiert und nicht erweiterbar. Dabei sind verschiedene primitive Typen für numerische und boolsche Werte sowie für Zeichen vorgesehen. Bei Referenztypen wird zwischen Klassen-, Schnittstellen- und Feldtypen unterschieden.

Durch die Deklaration von Klassen können Klassentypen geschaffen werden. Im obigem Beispiel 5.1 (vgl. Seite 22) wurde der Typ *Fahrzeug* geschaffen. Eine Variable, die von einem Klassentyp ist, kann über eine Referenz auf eine Instanz der entsprechenden Klasse verweisen. Beispielsweise verweist *standort* aus obigem Beispiel während der Laufzeit auf eine Instanz der Klasse *String*. Referenzen sind mit Zeigern anderer Programmiersprachen vergleichbar. Die Unterscheidung zwischen Instanzen und Referenzen ist für diese Arbeit nicht weiter relevant, so daß z.B. häufig auch von einer Variable mit einer Instanz vom Typ *Fahrzeug* gesprochen wird.

Schnittstellentypen sind den Klassentypen sehr ähnlich. Der Wertebereich eines Schnittstellentyps wird durch die Referenzen auf alle existierenden Instanzen der Klassen gebildet, welche die Schnittstelle implementieren.

Feldtypen werden über den Typkonstruktor [] erzeugt. Beispielsweise würde *String*[] *daten* eine Variable *daten* deklarieren, die ein Feld von Referenzen auf Instanzen vom Typ *String* aufnehmen kann. Felder werden immer über einen Basistyp gebildet, im Beispiel *String*. Über einen solchen Basistyp können auch mehrdimensionale Felder gebildet werden. Der Objektbegriff wird bei Java auf Felder ausgedehnt, d.h. neben Instanzen von Klassen sind auch Felder Objekte.

Es ist nicht zu erwarten, daß primitive Typen wesentliche Erkenntnisse über die Struktur einer Software liefern. Daher werden sie nicht in das Metamodell integriert. Klassen- und Schnittstellentypen werden, wie gezeigt, durch *Klasse* und *Schnittstelle* modelliert. Feldtypen werden bei der Definition der Benutzungsbeziehung (siehe Abschnitt 5.7) berücksichtigt.

5.4 Pakete und Importe

Zum Strukturieren von Java-Software können Entitäten in sogenannte Pakete eingeteilt werden. Die Angabe, zu welchem Paket eine Entität gehört, erfolgt am Anfang der Quelltextdatei, in der die Entität deklariert ist. Da in einer Quelltextdatei mehrere Entitäten deklariert werden können, gehören alle Entitäten einer Datei zum selben Paket.

Während einer Entität implizit alle Entitäten zur Verfügung stehen, die zum selben Paket gehören, sollten Entitäten anderer Pakete zunächst explizit importiert werden. Dies geschieht für einzelne Entitäten oder für ein gesamtes Paket über das Schlüsselwort *import*. Alle Entitäten einer Quelltextdatei importieren dieselben Entitäten und Pakete.

Nachfolgendes Beispiel zeigt einen Ausschnitt einer Quelltextdatei, deren Entitäten *Autovermierung* und *Tools* dem Paket *de.autoverm* zugeordnet werden. In den Entitäten der Datei stehen standardmäßig alle weiteren Entitäten des Paketes *de.autoverm* und aufgrund der beiden expliziten Importe alle Entitäten des Paketes *de.autoverm.reservierung* und die Entität *Fahrzeug* aus dem Paket *de.autoverm.inventar.fuhrpark* zur Verfügung. Weiterhin können auch ohne expliziten Import allen Entitäten des zu Java gehörenden Paketes *java.lang* verwendet werden, welches beispielsweise auch die bereits erwähnten Klassen *String* und *Object* umfaßt.

Beispiel 5.9: [Pakete und Importe]

```
package de.autoverm;
import de.autoverm.reservierung.*;
import de.autoverm.inventar.fuhrpark.Fahrzeug;

class Autovermietung { Fahrzeug[] fahrzeuge; ... }
class Tools { ... }
```

Pakete können untergeordnete Pakete besitzen. Hier wird von direkten und indirekten Sub- und Superpaketen gesprochen. Ähnlich zu Dateipfaden bei UNIX besteht der Name eines direkten Subpaketes aus dem des direkten Superpaketes, gefolgt von einem Trennzeichen und einem weiteren Bezeichner. Als Trennzeichen wird hier ein Punkt verwendet. Im obigem Beispiel ist das Paket *de.autoverm.reservierung* direktes Subpaket von *de.autoverm*. Das Paket *de.autoverm.inventar.fuhrpark* ist ebenfalls Subpaket von *de.autoverm*, aber kein direktes.

Der vollständig qualifizierende Name (VQN) einer Entität ist die Zusammensetzung ihres Entitätsnamens und des Namens ihres Paketes. Für die Entität *Fahrzeug* aus obigem Beispiel lautet er *de.autoverm.inventar.fuhrpark.Fahrzeug*. Mit Hilfe von voll qualifizierenden Namen kann der Import von Entitäten umgangen werden. Beispielsweise könnte die Variable *fahrzeuge* in *Autovermietung* aus Beispiel 5.9 auch wie folgt deklariert werden, wodurch der Import der Entität *Fahrzeug* entfallen könnte:

Beispiel 5.10: [Vewendung eines VQNs]

```
class Autovermietung {
  de.autoverm.inventar.fuhrpark.Fahrzeug[] fahrzeuge; ...
}
```

Die über das Schlüsselwort *package* erfolgende Paketangabe in einer Quelltextdatei ist optional. Erfolgt sie nicht, so werden die darin deklarierten Entitäten einem sogenannten anonymen Paket zugeordnet. Der VQN einer Entität aus dem anonymen Paket besteht einzig aus ihrem Entitätsnamen.

In das Metamodell werden Pakete über *Paket* integriert. Diese Spezialisierung von *Element* besitzt keine weiteren Attribute oder Assoziationen. Das von Element geerbte Attribut *name* beschreibt den Namen des Paket; beim anonymen Paket ist das Attribut mit dem leeren String („“) belegt. Wie zuvor festgestellt wurde, besitzen alle Elemente – egal welcher Art – einen VQN. Die Bedeutung von *name* wird daher dahingehend präzisiert, daß das Attribut immer den VQN eines Elements angibt.

Die Zugehörigkeit einer Entität zu einem Paket wird über *EP* modelliert (vgl. Abbildung 5.3). Dabei wird auf eine enthaltene Entität über die Assoziation *von* und auf das enthaltende Paket über die Assoziation *nach* verwiesen. Beide Assoziationen werden von *Beziehung* geerbt und sind nicht in der Abbildung enthalten. Über *PP* wird eine Hierarchisierung von Paketen abgebildet; *von* zeigt auf ein direktes Subpaket und *nach* auf das entsprechende direkte Superpaket. Auf *EE* wird im nachfolgenden Abschnitt eingegangen.

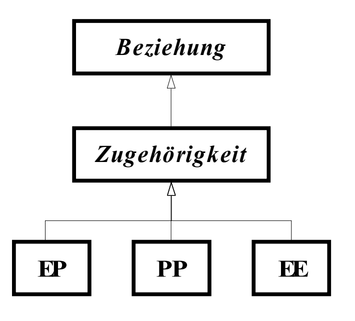


Abbildung 5.3: Arten von Zugehörigkeiten

Importe werden in Strukturmodellen wiederum durch eine Spezialisierung von *Verwendung* erfaßt. Diese heißt *Import* und besitzt keine weiteren Attribute oder Assoziationen. Für eine Instanz *imp* von *Import* gilt: *imp.von* verweist auf die importierende Entität und *imp.nach* auf das importiere Element.

5.5 Entitätsschachtelungen

Seit dem Vorliegen der Version 1.1 von Java können innerhalb der Deklaration von Klassen weitere Entitäten deklariert werden. Entitäten, die innerhalb einer Klasse deklariert sind, werden in der Sprachdefinition, auch wenn es sich um Schnittstellen handelt, als *Inner-Classes* bezeichnet. In dieser Arbeit wird zwischen inneren Klassen und inneren Schnittstellen unterschieden, bzw. wird, wo dies nicht notwendig ist, von inneren Entitäten gesprochen. Das Beispiel zeigt daneben weiterhin eine Sonderform innerer Entitäten, nämlich die sogenannten anonymen Klassen. Diese werden direkt bei der Instanziierung eines Objektes als Erweiterung einer bestehenden Klasse oder als Implementierung einer Schnittstelle deklariert. Im Beispiel wird die Klasse *Object* erweitert und deren Methode *hashCode* redefiniert.

Beispiel 5.11: [Innere Entitäten und anonyme Klassen]

```
package de.beispiele;
public class Außen {
  private class Innen { ... };
  Object h = new Object() { int hashCode() { return 0; } };
}
```

Der VQN einer inneren Entität ist die Zusammensetzung aus dem VQN der umschließenden Klasse und durch einem Punkt abgetrennte Name der inneren Entität. Der VQN von *Innen* aus obigen Beispiel ist demnach *de.beispiele.Außen.Innen*.

Innere Klassen können wiederum innere Entitäten enthalten, die dann ebenfalls innere Entität der äußersten Klasse sind, usw. An Stellen wo dies notwendig ist, wird in dieser Arbeit, analog zu direkten Subpakten und Subpaketen, zwischen direkten inneren Entitäten und (transitiven) inneren Entitäten unterschieden.

Die Schachtelung von Entitäten über die Spezialisierung *EE* von *Zugehörigkeit* modelliert (vgl. Abbildung 5.3). Dabei wird auf eine direkte innere Entität über die Assoziation *von* und auf die entsprechende direkte äußere Entität über die Assoziation *nach* verwiesen. Anonyme Klassen werden nicht in Strukturmodelle aufgenommen, da sie im Regelfall eine sehr geringe und nur lokale Relevanz haben.

5.6 Ausnahmebehandlung

Für Abweichungen vom normalem Programmablauf können in Java sogenannte Ausnahmen (*Exceptions*) oder Fehler (*Errors*) verwendet werden. Diese können innerhalb von Methoden aufgeworfen und behandelt werden. Bei Ausnahmen und Fehlern handelt es sich um Instan-

zen bestimmter Klassen. Klassen für Ausnahmen, im folgenden auch als Ausnahmeklassen bezeichnet, erben für gewöhnlich von der Klasse *Exception*, während Klassen für Fehler (Fehlerklassen) von der Klasse *Error* erben. Während Ausnahmeklassen für Abweichungen verwendet werden, auf die ein Programm reagieren können soll, werden mittels Fehlerklassen Abweichungen behandelt, die so schwerwiegend sind, daß sie normalerweise zu Terminierung eines Programmes führen. Neben Ausnahme- und Fehlerklassen können von einem Java-Programmierer auch weitere Klassen als aufwerfbar deklariert werden, wenn diese die Klasse *Throwable* erweitern.

Beispiel 5.12: [Ausnahmebehandlung]

```
(1) class Buchung {
(2)   void durchführen(Reservierung r) {
(3)     try{ Beleg b = sucheBeleg(); ... }
(4)     catch( BelegException e ) { System.out.println("Fehler!") }
(5)   }
(6)   Beleg sucheBeleg() throws BelegException {
(7)     Beleg b = ...
(8)     if( b.istFehlerhaft() ) throw new BelegException();
(9)     return b;
(10)  }
(11)}
```

Ausnahmen und auch Fehler werden durch das Schlüsselwort *throw* aufgeworfen (Zeile 8 im obigen Beispiel). Wenn Ausnahmen einer Klasse nicht innerhalb einer Methode behandelt werden, muß die Klasse i.d.R. im Kopf der Methoden als aufwerfbar deklariert werden (Zeile 6). Bestimmte Ausnahmeklassen, nämlich die, die von der Klasse *RuntimeException* erben, sowie auch die Fehlerklassen insgesamt, müssen nicht angegeben werden. Die Zeilen 3 und 4 zeigen die Behandlung von Ausnahmen der Klasse *BelegException* durch die Ausgabe eines Textes.

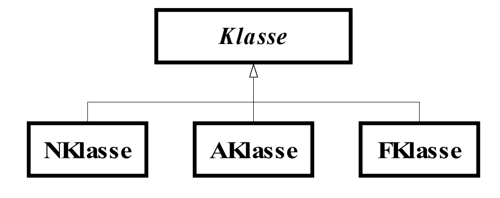


Abbildung 5.4: Spezialisierungen von „Klasse“

Im Metamodell werden die verschiedenen Arten von Klassen unterschieden, um den verschiedenen Einsatzgebieten Rechnung zu tragen. Es werden drei Spezialisierungen von *Klasse* gebildet: *NKlasse* für normale Klassen, *AKlasse* für Ausnahmeklassen und *FKlasse* für Fehlerklassen. Hierdurch erklärt sich, warum *Klasse* abstrakt ist (vgl. Abbildung 5.4).

5.7 Benutzungen zwischen Entitäten

Java-Software wird insbesondere auch dadurch strukturiert, daß Entitäten andere Entitäten benutzten um eine Aufgabe zu erfüllen. So kann eine Entität beispielsweise Klassen- oder Instanzvariablen vom Typ einer anderen Entität deklarieren oder Methoden deklarieren, die einen Wert vom Typ einer anderen Entität als Argument erwarten. Im folgenden wird die Frage geklärt, wann im Rahmen dieser Arbeit davon gesprochen wird, daß eine Entität eine andere Entität benutzt. Zudem werden verschiedene Eigenschaften von Benutzungen identifiziert.

Es können zunächst die folgenden Formen von Benutzungen einer Entität *Used* durch eine Entität *User* unterschieden werden:

1) *Benutzung zur Variablendeklaration*. In *User* wird eine Variable vom Typ *Used* oder eines Feldtyps über *Used* deklariert.
2) *Benutzung zur Konstruktordeklaration*. Im Kopfbereich eines Konstruktors wird *Used* oder ein Feldtyp über *Used* aufgeführt. Möglich ist die Nennung als ein formaler Parametertyp oder als aufwerfbarer Typ.
3) *Benutzung zur Methodendeklaration*. Im Kopfbereich einer Methode wird *Used* oder ein Feldtyp über *Used* aufgeführt. Hier ist die Nennung als Rückgabetyp, als ein formaler Parametertyp oder als aufwerfbarer Typ möglich.
4) *Benutzung zur Implementierung*. Im Codeblock einer Methode, eines Konstruktors oder eines Initialisieres oder innerhalb eines Initialisierungsausdrucks für eine Variable wird eine Instanz von *Used* erzeugt, eine lokale Variable vom Typ *Used* deklariert oder auf eine Methode oder eine Variable vom Typ *Used* zugegriffen bzw. dies geschieht mit einem Feldtyp über *Used*.

Gilt eine diese vier Bedingung für einen Feldtyp über *Used* wird von einer *Benutzung (zur Variablendeklaration, zur Konstruktordeklaration, ...) als Feld* gesprochen. Für die ersten drei Fälle kann festgehalten werden, welchen Zugriffsmodus die deklarierte Variable, der deklarierte Konstruktor oder die deklarierte Methode besitzt. Hierdurch wird erkennbar, inwieweit die Benutzung außerhalb von *Used* für Entitäten, die selbst wiederum *User* benutzen, sichtbar ist. Dementsprechend kann allen Benutzungen zur Implementierung der Zugriffsmodus *private* zugeordnet werden, da sie von außen nicht sichtbar sind.

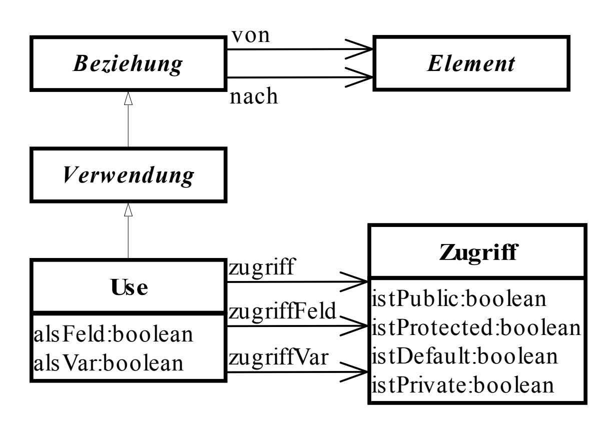


Abbildung 5.5: Modellierung von Benutzungen

Vielfach werden zwischen zwei Entitäten *User* und *Used* mehrere und auch verschiedenartige Benutzungen gleichzeitig vorliegen. Beispielsweise werden innerhalb des Rumpfes einer Methode von *User*, die einen Parameter vom formalem Typ *Used* erwartet, häufig auch Methoden von *Used* aufgerufen, so daß eine Benutzung zur Methodendeklaration und mindestens eine Benutzung zur Implementierung vorliegen. Da es wenig Sinn macht, alle diese Benutzungen zu visualisieren, wird bereits im Strukturmodell einer darzustellenden Software eine Verdichtung vorgenommen. Das Vorliegen einer oder mehrerer Benutzungen einer Entität *Used* durch eine Entität *User* wird daher durch eine *Use*-Instanz *use* modelliert. Bei dieser verweist *use.von* auf die Modellierung von *User* und *use.nach* auf die Modellierung von *U-sed*. Über Instanzen der Klasse *Use* werden verschiedene Eigenschaften der vorhandenen Benutzungen modelliert (vgl. auch Abbildung 5.5). Diese werden in Tabelle 5.2 auf nachfolgender Seite für *use* beschrieben.

Modellierung	Semantik
use.zugriff	Höchster Zugriffsmodus, der bei den Benutzungen von *Used* durch *User* auftritt.
use.alsFeld	Die Variable ist genau dann mit *true* belegt, wenn mindestens eine Benutzung als Feld vorliegt. Welche der vier Formen diese Benutzung hat, ist dabei unerheblich.
use.alsVar	Die Variable ist genau dann mit *true* belegt, wenn mindestens eine Benutzung zur Variablendeklaration vorliegt. Dies ist unabhängig davon, ob es sich dabei um eine Benutzung als Feld handelt oder nicht.
use.zugriffFeld	Höchster Zugriffsmodus, der bei den Benutzungen von *Used* durch *User* als Feld auftritt. Nur definiert, wenn *use.e.alsFeld* mit *true* belegt ist.
use.zugriffVar	Höchster Zugriffsmodus, der bei den Benutzungen von *Used* durch *User* zur Variablendeklaration auftritt. Nur definiert, wenn *use.e.alsVar* mit *true* belegt ist.

Tabelle 5.2:Eigenschaften von Benutzungen

Die besondere Kennzeichnung von Benutzungen als Feld und Benutzungen zur Variablendeklaration erfolgt in Anlehnung an viele Modellierungsprachen, wie z.B. der UML, bei denen Assoziationen mit einer Multiplizität von *n* (im Gegensatz zu einer Multiplizität von 1) und Aggregationen gesondert markiert werden können.

5.8 Zusammenfassung

Abbildung 5.6 zeigt das in diesem Kapitel entwickelte Metamodell für Strukturmodelle. Um die Abbildung etwas übersichtlicher zu gestalten, ist *Entität* zweifach aufgeführt.

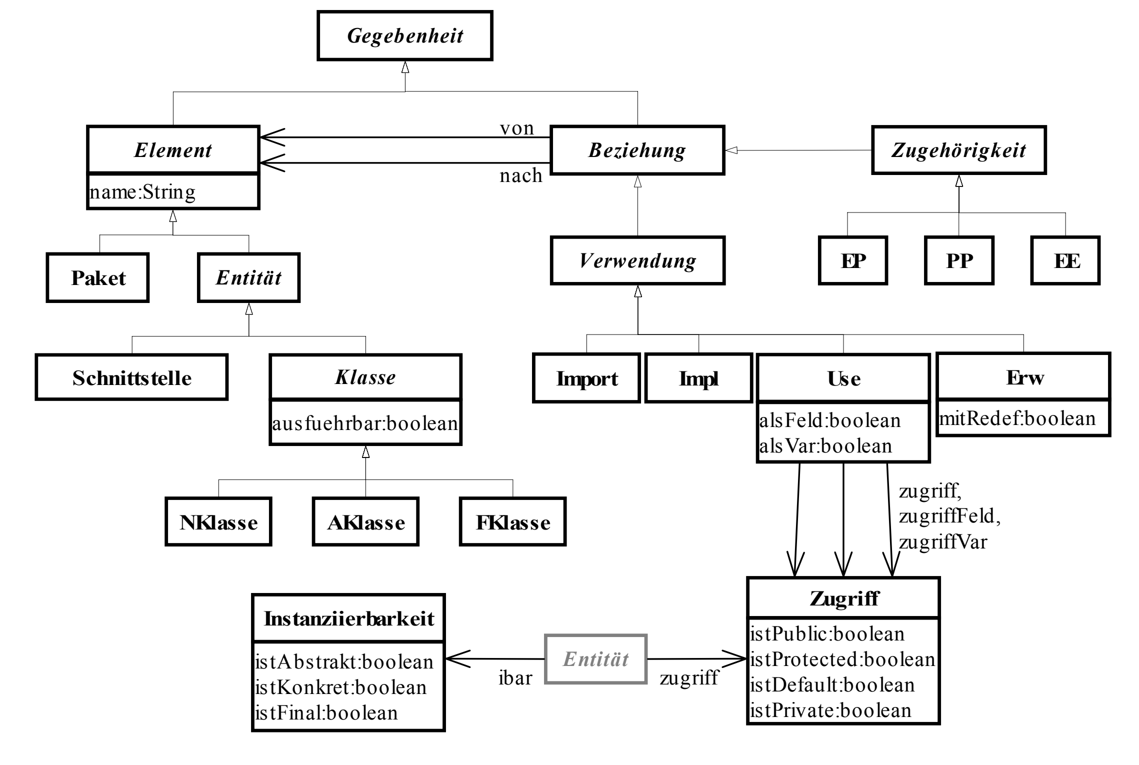


Abbildung 5.6: Metamodell für Strukturmodelle

6 Verwendete Softwarebeispiele

Als ein Schritt der Konzeption galt es, Beispiele für Java-Software mit nicht trivialem Umfang zu finden, anhand derer ein Konzept zur Visualisierung entwickelt werden konnte. Dabei stellte sich heraus, daß der Quelltext derartiger Software kaum frei verfügbar ist. Das Verwenden von Quelltexten bietet sich aber aus praktischen Gründen als Basis für das Erzeugen von Visualisierung gegenüber Alternativen, wie z.B. das Benutzen des Bytekodes einer Software, an. So wurde als Softwarebeispiel vor allem die Java-API in der Version 1.1.8 herangezogen. Im Verlauf der Arbeit konnte auch zunehmend die bis dahin erstellte Implementierung verwendet werden. Zusätzlich wurde ein am Lehrstuhl für Software-technologie des Fachbereich Informatik der Universität Dortmund erstelltes System zur multimedialen Präsentation des Altenberger Doms (AD1300) (vgl. z.B. [Alf99]) als Beispiel genutzt. Tabelle 6.1 nennt einige Daten über die Größe der Beispiele. In der Tabelle findet sich die während dieser Arbeit erstellte Implementierung unter dem Namen *J3Browser*.

	Java-API	**J3Browser**	**AD1300**
Quelltextgröße			
Quelltextdateien	679	190	82
Quelltextzeilen[1]	150289	39797	14712
Elemente (gesamt)	**767**	**260**	**98**
Klassen / Schnittstellen / Pakete	609 / 134 / 24	232 / 17 / 11	80 / 9 / 9
Beziehungen (gesamt)	**5278**	**1476**	**422**
Zugehörigkeiten (gesamt)	762	289	99
PP/EP/EE	12 / 743 / 7	10 / 249 / 30	5 / 89 / 5
Verwendungen (gesamt)	4516	1187	330
Use/Erw/ Impl/Import	2243 / 479 / 178 / 1616	611 / 116 / 40 / 420	132 / 29 / 10 / 159
Sonstige Daten			
max. Klassenhierarchiegröße[2]	144	27	22
max. Schnittstellenhierarchiegröße[2,3]	19	1	1
max. Tiefe Entitätsschachtelungen[4]	1	1	1
Tiefe Pakethierarchie[5]	3	4	4

Tabelle 6.1: Daten zu den verwendeten Softwarebeispielen

1) Gezählt inklusive aller Kommentare und Leerzeilen.

2) Eine Menge durch Erweiterungsbeziehungen zusammenhängender Entitäten bildet eine Hierarchie. Angegeben ist die Anzahl der Entitäten in der Hierarchie, welche die meisten Entitäten umfaßt. Implizite Erweiterungen der Klasse "java.lang.Object" wurden dabei nicht berücksichtigt.

3) In den Softwarebeispielen "J3Browser" und "AD1300" existieren keine Erweiterungsbeziehungen zwischen Schnittstellen, so daß alle Schnittstellenhierarchien hier nur aus jeweils einer Schnittstelle bestehen.

4) In keinem der Beispiele hat eine innere Klasse selbst innere Entitäten.

5) Anzahl der spezifischen Bestanteile des vollständig qualifizierenden Namens des tiefsten Pakets in der Hierarchie.
Beispiel: *java.awt.event* => 3 Bestandteile. *de.j3browser.darstellung.diagramm.ausrichtung* => 4 Bestandteile, da *de* nicht spezifisch für den J3Browser ist.

7 Einige Visualisierungstechniken und -systeme

Im ersten Teil dieser Arbeit wurde bereits das SemNet-Projekt angesprochen. Es existieren eine Reihe von weiteren Visualisierungstechniken und -systemen. Diese bieten sich als Ausgangspunkt für eigene Überlegungen zur Antwort auf die zweite konzeptionelle Frage „*Wie sollen Softwarestrukturvisualisierungen aufgebaut sein?*“ an. Einige interessante Arbeiten werden daher im folgenden vorgestellt. Dies erfolgt zweigeteilt zunächst für Visualisierungstechniken und dann für Visualisierungssysteme. Für eine ausführliche und auch andere Techniken und Systeme umfassende Vorstellung vgl. [You96], [Eng95] oder [Wün97].

7.1 Visualisierungstechniken

7.1.1 Semiotische Prinzipen nach Franck und Ware

Franck und Ware untersuchten Möglichkeiten, wie Netzwerke aus Knoten und Kanten in drei Dimensionen geeignet dargestellt werden können [FW94]. Dabei betrachteten sie konkret die Darstellung von C++ Programmen. Sie zeigen sechs sogenannte semiotische Prinzipen auf, die im folgenden vorgestellt werden.

Körperförmige Elementdarstellung. Darzustellende Elemente, wie z.B. die Klassen einer objektorientierten Programmes, sollen als Objekte dargestellt werden. Mit *Objekten* sind dabei graphische Körper wie beispielsweise Quader, Kugeln etc. gemeint. Obwohl in der Arbeit von Franck und Ware der Begriff Objekt statt Körper verwendet wird, wird nachfolgend von Körpern oder auch (dreidimensionalen) Symbolen gesprochen, da sich dadurch eine klare Trennung zum eingeführten allgemeinen Begriff des graphischen Objektes ergibt. Durch die körperhafte Form wird nach den Aussagen von Franck und Ware die Erkennbarkeit und die Einprägsamkeit der Darstellung verbessert.

Die körperliche Darstellung von Elementen im dreidimensionalen Raum kann auch als Übertragung deren flächiger Darstellung im zweidimensionalem Raum gesehen werden. Beispielsweise werden in Klassendiagrammen z.B. Rechtecke zur Darstellung von Klassen verwendet. Die Verwendung anderer, bspw. linienförmiger, Darstellungen für Elemente ist weit weniger üblich.

Pfeilförmige Beziehungsdarstellung. Beziehungen zwischen darzustellenden Elementen sollen als Pfeile dargestellt werden, wobei Franck und Ware diese Aussage auf die gegenseitige Verwendung der Elemente oder der Kommunikation zwischen diesen beziehen. Dies ist eine direkte Übertragung der pfeilförmigen Beziehungsdarstellung im zweidimensionalen Raum. Wie bereits erwähnt, ergeben sich im dreidimensionalen Raum aber aufgrund wechselnder Betrachtungspositionen und -winkel erweiterte Anforderungen an die Gestaltung. Hier stellt sich vor allem das Problem, die Richtung von Pfeilen erkennbar zu machen. Franck und Ware haben auch diesen Aspekt betrachtet, und treffen die Aussage, daß insbesondere durch Farbwechsel innerhalb des Pfeils, die Erkennbarkeit der Richtung unterstützt werden kann.

Naturgemäß werden Pfeile in der dreidimensionalen Darstellung ebenfalls durch Körper, wie z.B. Zylinder, realisiert. Diese Körperlichkeit steht aber gegenüber der oben geforderten nur im Hintergrund, ist also ein Realisierungsaspekt gegenüber einer geforderten Eigenschaft im Fall des Prinzips der körperförmigen Elementdarstellung.

Eigenschaftsdarstellung durch graphische Eigenschaften. Dargestellte Elemente und auch Beziehungen können über verschiedene Eigenschaften verfügen. Handelt es sich bei einem dargestellten Element beispielsweise um eine Klasse, kann eine ihrer Eigenschaften der

Umfang des zu ihrer Implementierung notwendigen Quelltextes sein. Das dritte semiotische Prinzip von Franck und Ware sagt aus, daß derartige Eigenschaften eines Sachverhaltes über die Oberflächeneigenschaften oder die Form des entsprechenden graphischen Objektes anzuzeigen sind. Bei Eigenschaften von Oberflächen handelt es sich beispielsweise um deren Farbe oder Transparenz. Unter Eigenschaften der Form können neben der allgemeinen Gestaltung des graphischen Objektes auch dessen Größe verstanden werden, die sich z.B. dafür eignen könnte, den Quelltextumfang aus obigem Beispiel darzustellen.

Orthogonalität. Voneinander unabhängige Eigenschaften eines darzustellenden Sachverhalts sollen durch unabhängige Eigenschaften des entsprechenden graphischen Objekts dargestellt werden. Weiterhin sollten die zur Darstellung gewählten Eigenschaften des graphischen Objektes unabhängig von äußeren Gegebenheiten sein. Hierfür geben Franck und Ware zwei Beispiele. Ein Verstoß gegen dieses Prinzip wäre es beispielsweise, wenn man die Helligkeit der Färbung eines graphischen Objektes zur Darstellung einer Eigenschaft verwendet und gleichzeitig eine Simulation von wechselnder Beleuchtung erfolgt. Da sich durch die wechselnde Beleuchtung die Helligkeit der Färbung ebenfalls verändert, ist die zweifelsfreie Erkennbarkeit der Eigenschaft nicht gegeben. Eine andere Form des Verstoßes gegen das Orthogonalitätsprinzip kann auftreten, wenn beispielsweise eine Eigenschaft eines Elementes dadurch dargestellt würde, daß das entsprechende Objekt entweder als Drahtmodell eines Körper oder als kompletter Körper dargestellt würde. Würde dann eine andere Eigenschaft durch die Farbe des Körpers dargestellt, so würde sich die Situation ergeben, daß bei der Darstellung des Körpers als Drahtmodell die Farbe kaum zu erkennen wäre.

Objektkonstanz. Wenn sich die Darstellung eines grafischen Objektes während der Betrachtung eines Diagramms verändern kann, z.B. weil sich der Blickwinkel des Betrachters auf das Objekt verändert, so sollte diese Veränderung so gestaltet werden, daß es erkennbar bleibt, daß es sich weiterhin um *dasselbe* Objekt handelt. Diese Eigenschaft wird als *Objektkonstanz* bezeichnet.

Um die Objektkonstanz bei Veränderungen des Blickwinkels zu erhalten, kann das Objekt z.B. so entworfen werden, daß es symmetrisch bezüglich einer vertikalen Achse ist. Ist es mit Beschriftungen versehen, so können diese bei Änderungen des Blickwinkels so bewegt werden, daß sie lesbar bleiben. Besitzt der darzustellende Sachverhalt mehrere Zustände, die graphisch unterschieden werden, so kann bei einem Zustandswechsel Objektkonstanz durch animierte Übergänge bewahrt bleiben.

Rekursivität. Bei einem rekursiven Aufbau des darzustellendem Sachverhalt sollten auch dessen Darstellung in graphischer Hinsicht rekursiv sein. Ein gutes Beispiel hierfür ist die Zusammensetzung von Java-Software aus Paketen, die neben Entitäten auch untergeordnete Pakete besitzen können und so weiter. Die Visualisierung dieses Aufbaus soll nach Franck und Ware ebenfalls eine rekursive Struktur besitzen, die z.B. dadurch erzielt werden kann, daß Symbole für untergeordnete Pakete in Symbole für übergeordnete Pakete geschachtelt werden, wie dies auch gängige Praxis ist. Weiterhin sollten dann gleiche Arten von Beziehungen zwischen den inneren oder den äußeren Paketen oder Entitäten mit den gleichen Mechanismen dargestellt werden, also z.B. durch gleichartige gestaltet Pfeile.

7.1.2 Einfache Darstellungsformen für Hierarchien

Die ***baumförmige Darstellung*** hierarchischer Strukturen ist weit verbreitet und kann als allgemein bekannt vorausgesetzt werden. Sie soll hier nur kurz im Beispiel gezeigt werden (vgl. Abbildung 7.1a auf nachfolgender Seite).

Im folgenden wird von einer *baumförmigen Hierarchie* gesprochen, wenn diese sich durch einen Baum darstellen läßt. Dies ist der Fall, wenn für jeden Knoten, mit Ausnahme der Wurzel, genau ein übergeordneter Knoten vorhanden ist. Die Wurzel ist der Knoten der

Hierarchie, der keinen übergeordneten Knoten besitzt.

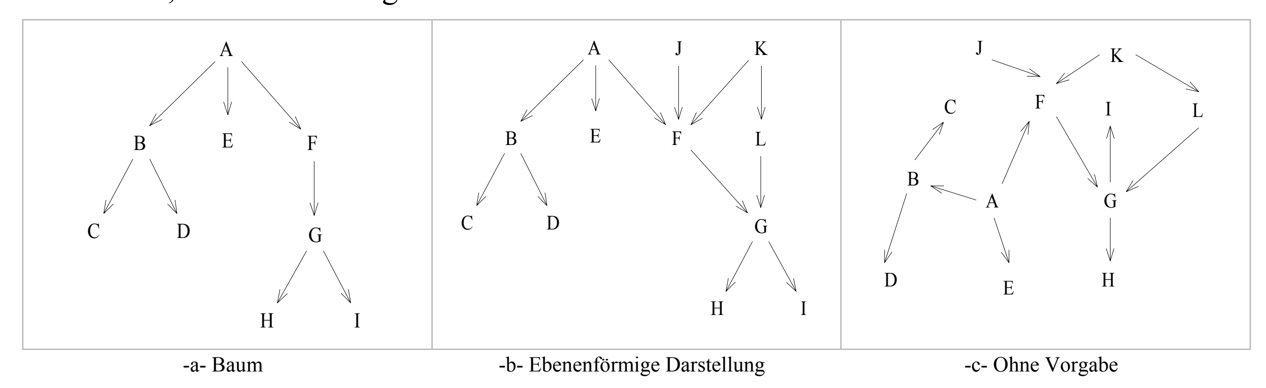


Abbildung 7.1: Einfache Darstellungsformen für verschiedenartige Hierarchien

Bei Bäumen werden gemäß der Hierarchiestufen äquidistante Ebenen in der Darstellung eingeführt. Für diese Ebenen werden dann weitere Anordnungsvorgaben gemacht. Letztere Vorgaben können aufgehoben werden, so daß die Knoten innerhalb der Ebenen flexibel plazierbar werden. Zudem können so auch Hierarchien ohne Baumförmigkeit dargestellt werden (vgl. Abbildung 7.1b). Eine solche Darstellung wird als ***ebenenförmig*** bezeichnet.

Schließlich kann auch die Aufteilung in Ebenen aufgegeben werden. Weniger restriktiv kann zwar gefordert werden, daß in der Hierarchie übergeordnete Knoten oberhalb von untergeordneten plaziert werden müssen, ohne das allerdings ein Abstand vorgegeben wird. Dies wird nachfolgend als ***top-down-Darstellung*** betitelt. Bei Aufgabe auch der letzten Forderung können Hierarchien nur noch schwer erkannt werden, wenn dieses Gebot nicht durch im Prinzip gleichwertige Forderungen, wie z.B. übergeordnete Knoten sind links von untergeordneten Knoten anzuordnen, ersetzt wird (vgl. Abbildung 7.1c).

7.1.3 Cone Trees und Cam Trees

Die namensgebenden Kegel (*Cones*) spielen eine wichtige Rollen bei der durch Robertson, Mackinlay und Card eingeführte Cone Tree-Darstellung [RMC91, RCM93]. Diese dienen zur Darstellung baumförmiger Hierarchien. Die Wurzel eines jeden Teilbaums der Hierarchie bildet die Spitze eines semitransparenten Kegels. Die der Wurzel direkt untergeordneten Knoten werden auf einem Kreis plaziert, der die Grundfläche des Kegels bilden. Diese Konstruktion wird rekursiv fortgesetzt, so daß eine Struktur entsteht, wie sie in Abbildung 7.2 gezeigt wird. Zeigt man die Hierarchie nicht vertikal wie in der Abbildung, sondern horizontal, dann wird von *Cam Trees* gesprochen.

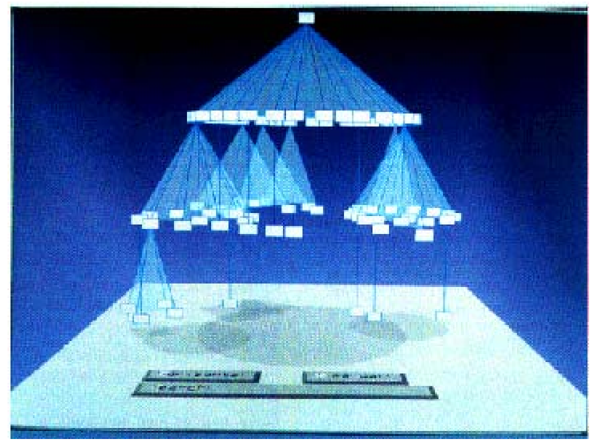


Abbildung 7.2: Cone Tree (Quelle: [RMC91])

Robertson, Mackinlay und Card treffen die Aussage, daß sich mit Hilfe von Cone Trees Strukturen visualisieren lassen, die bis zu tausend Knoten umfassen [RMC91]. Ein wichtiges Hilfsmittel dabei ist die Animation der Darstellung, die es dem Benutzer erlaubt, einzelne Teilbäume eines Cone Tree zu drehen. Dadurch kommen zuvor perspektivisch verkleinert Knoten aus dem Hintergrund, in dem sie dargestellt wurden, um einen Kontext zum aktuellen Vordergrunds zu liefern.

Würde die Drehung ohne die Darstellung von Zwischenschritten alleine durch Umschalten

zwischen einer aktuellen und einer gewünschten Ansicht durchgeführt, so würde aus dieser Verletzung des Prinzip der Objektkonstanz folgen, daß ein Betrachter jedesmal eine gewisse Zeit benötigt, sich wieder in der Darstellung zurecht zu finden. Durch die animierte Darstellung wird diese Zeit minimiert.

Ein Nachteil der Cone Trees ist es, daß ihre Gesamtgröße exponentiell mit der Tiefe der Struktur, d.h. dem maximalen Abstand zwischen ihrer Wurzel und den Blätter, wächst. Zu diesem exponentiellen Größenwachstum kommt es auch bei der zuvor betrachteten baumförmigen Darstellung. Dort kann dieses Wachstum aber als störender empfunden werden, da es nicht auf drei, sondern nur auf zwei Dimensionen verteilt, erfolgt. Als eine Abhilfe werden neben den sog. *Information Cubes* (vgl. nachfolgender Abschnitt) sogenannte *Fractal Views* von Koike und Yoshihara vorgeschlagen [KY93]. Auf die Vorstellung derartiger Verfahren wird aber verzichtet, da die Softwarebeispiele gezeigt haben, daß Cone Trees (ggf. zusammen mit Information Cubes) i.d.R. ausreichend sind. Größere Hierarchien, deren Visualisierung lohnt und die dazu nicht geeignet aufgespaltet werden können, dürften nur bei außergewöhnlich großen Softwareprojekten zu erwarten sein.

7.1.4 Information Cubes

Information Cubes werden ebenfalls für baumförmige Hierarchien verwendet. Sie wurden von Rekimoto vorgestellt [Rek93]. Mit Hilfe einer geschachtelten Darstellung von Knoten können umfangreiche Hierarchien mit über tausend Knoten dargestellt werden. Abbildung 7.3 zeigt ein Beispiel.

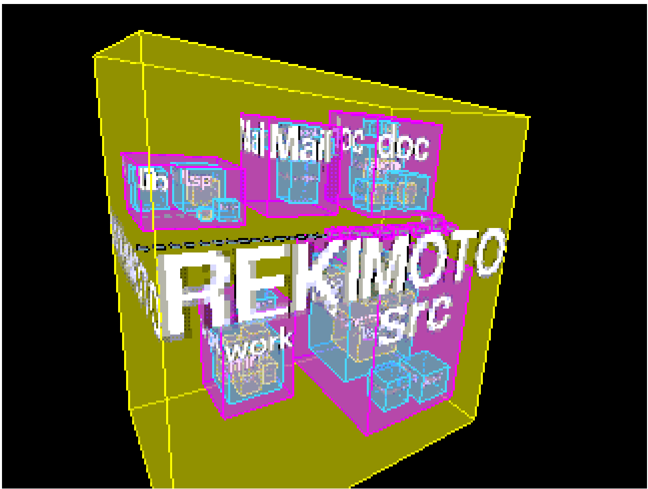


Abbildung 7.3: Information Cubes (Quelle: [You96])

Der Wurzelknoten wird als ein Würfel (*Cube*) dargestellt, der die gesamte Darstellung umfaßt. An der Stirnseite kann er beschriftet werden (in der Abbildung *REKIMOTO*). In dem Würfel werden geschachtelt kleinere Würfel überlappungsfrei dargestellt, welche die der Wurzel direkt untergeordneten Knoten repräsentieren. Diese können ebenfalls beschriftet werden und enthalten wiederum die Darstellung von Würfeln für ihre direkten Nachfolger in der Hierarchie usw. Durch eine Semitransparenz der Würfelwände wird zum einen erreicht, daß geschachtelte Würfel erkennbar bleiben. Zum anderen ergibt sich durch die Addition der Transparenzwerte aber auch eine Ausblendung von Würfeln in tiefen Schachtelungsebenen, wodurch die Komplexität der Darstellung reduziert wird. Um auch Würfel in tiefen Schachtelungsebenen sichtbar zu machen, sind Navigationsmöglichkeiten notwendig, durch welche die Würfel „betreten“ werden können.

7.1.5 Information Landscapes

Die dreidimensionale Visualisierungstechnik der Information Landscapes nutzt die Metapher

der Landschaft. Dem Betrachter wird ein Bild präsentiert, das an das Aussehen einer natürlichen Landschaft erinnern soll (vgl. Abbildung 7.4).

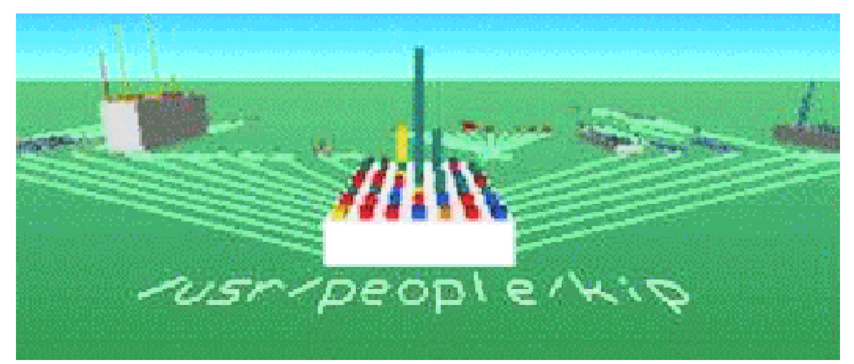


Abbildung 7.4: Visualisierung eines Dateisystem durch den FSN. (Quelle: [Wün97])

Die Abbildung entstammt dem File System Navigator (FSN) von SGI [TS92]. Er visualisiert die Struktur von Dateisystemen unter UNIX. Dabei bilden die Verzeichnisse Blöcke auf dem Boden der Darstellung. Sie erinnern an Gebäude in einer Landschaft. Die einzelnen Dateien der Verzeichnisse werden durch Quader auf den Blöcken dargestellt. Die Quader unterscheiden sich in Volumen und Farbe, um verschiedene Dateiattribute, wie Größe und Zugriffsrechte, zu verdeutlichen. Die Beziehung zwischen über- und untergeordneten Verzeichnissen wird durch Verbindungslinien dargestellt, die – entsprechend der Landschaftsmetapher – an Straßen zwischen den Gebäuden erinnern. Der Benutzer von FSN kann durch die Landschaft navigieren, indem er z.B. virtuell über sie „hinweg fliegt“.

Problematisch an Information Landscapes ist meiner Ansicht nach die geringe Ausnutzung des verfügbaren dreidimensionalen Raumes. Der „Himmel“ bleibt ungenutzt. Insbesondere wird es bei der Darstellung nicht planarer Graphen Überschneidungen wie im Zweidimensionalem geben, wenn – wie in der Abbildung – Verbindungslinien nur auf der Bodenfläche gezeichnet werden.

Eingesetzt werden Information Landscapes auch im Harmony Hyper-G Browser [And95]. Hier findet sich auch eine Nutzung des Himmels zur Darstellung weiterer Beziehungen [APW96].

7.1.6 Dreidimensionaler hyperbolischer Raum

Für die Darstellung besonders umfangreicher Graphen wurden Darstellungstechniken für den dreidimensionalen Raum entwickelt, die auf einer Projektion auf Kugeln beruhen und eine gute Berücksichtigung des Kontext und Fokus Problems sowie eine einfache Navigation bieten. Es werden im Wesentlichen drei Modelle unterschieden [Wün97] (vgl. auch Abbildung 7.5 auf nachfolgender Seite):

Beim ***Projective oder Klein Modell*** wird der darzustellende Graph auf die Innenfläche einer Kugel projiziert (vgl. Abbildung 7.5a).

Das ***Conformal oder Poinncaré Modell*** ist eine Erweiterung des Projective Modells. Knoten werden als Teile der Kugeloberfläche dargestellt, die durch bogenförmige Kanten verbunden werden (vgl. Abbildung 7.5b).

Inside Modell: Der Betrachtungsstandort liegt innerhalb der Kugel, die nicht verlassen werden kann (vgl. Abbildung 7.5c).

Die Navigation kann in den drei Fällen bspw. dadurch erfolgen, daß die Kugel durch den Betrachter gedreht werden kann und so verschiedene Knoten in den Vordergrund gelangen.

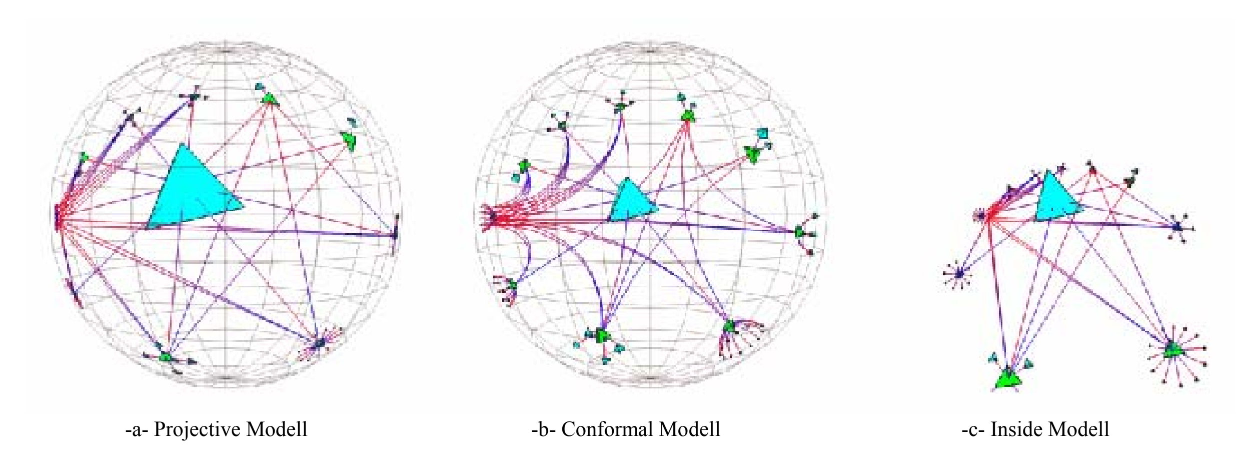


Abbildung 7.5: Modelle im dreidimensionaler hyperbolischer Raum (Quelle: [Wün97])

7.1.7 Graph Drawing

Graph Drawing ist ein Forschungsbereich, der sich mit der Entwicklung und Bewertung von Algorithmen beschäftigt, die ansprechende und verständliche Darstellungen von Graphen erzeugen sollen. Einen Überblick über diesen Bereich liefert [BET+94]. Es finden sich vor allem folgende drei Vorgehensweisen, um Darstellungen zu berechnen: die Anwendung spezifischer Algorithmen für bestimmte Sonderformen von Graphen, wie z.B. Bäume, sowie die Simulation von Federmodellen und die Optimierung von Kostenfunktionen für allgemeine Graphen.

Die Verwendung des Federmodells erfolgt, indem Knoten als Verbindungspunkte interpretiert werden, zwischen denen verschieden dimensionierte Federn „gespannt" werden. Wichtige Merkmale der Dimensionierung einer Feder sind ihre Federkonstante und ihre Ruhelänge, d.h. die Länge, welche die Feder annimmt, wenn keine externen Kräfte auf sie einwirken. Federn besitzen eine zur Federkonstante und ihrer aktuellen Dehnung bzw. Stauchung proportionale potentielle Energie. Möchte man z.B., daß sich Knoten nicht überschneiden, so plaziert man zwischen je zwei Verbindungspunkten eine Feder mit einer Ruhelänge, die größer ist als die Radien der Knoten und wählt eine große Federkonstante. Weiterhin sorgen Federn zwischen jeweils zwei Knoten, die mit einer Kante verbunden sind dafür, daß die Knoten aufeinander zustreben und unverbundene Feder sich voneinander entfernen. Ist ein solches System modelliert, wird die durch die Federkräfte resultierende Bewegung der Knoten simuliert, bis die Summe aller wirkenden Kräfte einen gesetzten Schwellenwert unterschreitet. Man erhofft sich, daß das Ergebnis dieser Simulation eine ansprechende Darstellung liefert. Tatsächlich wurde dieser Ansatz auch in verschiedenen Systemen mit Erfolg implementiert [FR91, BF96, RMS97]. Ein Beispiel für eine über ein Federmodell erzeugte Darstellung wurde bereits in Abbildung 2.4 auf Seite 6 gegeben.

Bei einem Vorgehen mittels der Optimierung einer Kostenfunktion ist diese zunächst zu definieren, wobei hier versucht werden muß, quantifizierbare Eigenschaften von Darstellungen zu finden, die deren Verständlichkeit bestimmen. Beispiele hierfür könnten die Anzahl der Kantenüberschneidungen oder auch die durchschnittliche Entfernung zwischen den Knoten sein. Zur Optimierung der Kostenfunktion wird relativ häufig das Simulated Annealing Verfahren eingesetzt [MRS95, CT96, DH96].

7.1.8 Fish Eye Views

Bei der Darstellung von Strukturen ist es überaus wichtig, dem Betrachter detaillierte Informationen zu liefern und gleichzeitig die Einbettung dieser Details in den Gesamtzusammenhang zu vermitteln. Es wird dabei i.d.R. nicht möglich sein, alle Details der Struktur gleichzeitig und gleichberechtigt darzustellen, da die Darstellung dann zu komplex

würde. Somit ist eine Auswahl zu treffen. In diesem Zusammenhang wird vom Fokus und Kontextproblem gesprochen, zu dessen Lösung in der Literatur mehrere Ansätze vorgeschlagen werden.

Ein Ansatz stellen die Fish Eye Views (FEV) dar, die auf Furnas zurück gehen [Fur81, nach Wün97]. Sie lassen sich unterteilen in Filter-FEV und Verzerrungs- oder Distortions-FEVs. Verzerrungs-FEVs simulieren die Optik von Fischaugen: Objekte in der Mitte des Sichtfeldes, dem Fokus, werden vergrößert dargestellt, während Objekte am Rand stark verkleinert werden und dadurch mehr Kontext darstellbar wird. Ein Beispiel für eine Umsetzung ist das Hyperbolic Lens System [LRP95].

Filter-FEVs hingegen variieren die Menge der dargestellten Objekte abhängig von einem Brennpunkt, d.h. eines selektierten Objektes. Für alle andere Objekte wird ein Wert - der Degree of Interest (DOI) - bestimmt. Der DOI eines Objektes ist dabei abhängig von der globalen Relevanz des Objektes - z.B. ist bei der Darstellung eines Baumes die Wurzel global relevanter als ein Blatt - und dem Abstand des Objektes zum Brennpunkt, gemessen z.B. anhand des Gewichtes verbindender Kanten. Liegt der DOI eines Objektes nun unterhalb eines festgelegten Schwellenwertes, so wird die Darstellung des Objektes unterdrückt. Filter-FEV lassen sich gut mit anderen Visualisierungstechniken kombinieren, da sie nicht eine bestimmte Anordnung der Darstellung erfordern.

7.1.9 Weitere Techniken für das Fokus- und Kontextproblem

In [PFW98] werden weitere Techniken zur Behandlung des Fokus- und Kontextproblems vorgestellt. Die dort zu findenden Ausführungen sollen hier kurz zusammengefaßt werden.

Rapid Zooming Technik. Bei der Rapid Zooming Technik erfolgt eine benutzergesteuerte Vergrößerung bzw. Verkleinerung des Darstellungsmaßstabes. Wichtig hierbei ist ein fließender Übergang zwischen den Maßstäben, so daß die Darstellung vom Benutzer in einen Gesamtzusammenhang integriert werden kann.

Sowohl die Rapid Zooming Technik als auch die im letzten Abschnitt besprochene Verzerrungstechnik sind bei einer dreidimensionalen Darstellung auf natürliche Weise integriert: eine Verzerrung ergibt sich durch eine perspektivische Darstellung, durch welche die Objekte im Vordergrund gegenüber denen im Hintergrund vergrößert erscheinen. Bei der Annäherung an Objekte ergibt sich zudem deren Vergrößerung und somit ein Rapid Zooming. Problematisch ist es allerdings, daß Objekte im Hintergrund nicht notwendigerweise zu denen im Vordergrund im Zusammenhang stehen. Für die Plazierung von graphischen Objekten wird zwar i.a. gefordert, daß zusammenhängende Objekte nahe beieinander liegen, trotzdem kann es in gewisser Weise willkürlich sein, welche Objekte im Hintergrund erscheinen. Dies ist insbesondere auch deshalb so, weil eine Betrachtung aus unterschiedlichen Winkeln erfolgen kann, wobei sich der sichtbare Hintergrund verändert.

Elisionstechnik. Bei der Elisions- oder Auslassungstechnik werden Elemente der Darstellung im Kontext vereinfacht dargestellt, während im Fokus eine detaillierte Darstellung erfolgt.

Multiple Bildschirmfenster. Häufig findet man die Verwendung zweier Bildschirmfenster, die praktisch den gleichen Sachverhalt anzeigen. Während in einem aber eine grobe Übersicht über den Gesamtzusammenhang gegeben wird, erfolgt im anderem Fenster eine detaillierte Darstellung eines kleinen Ausschnittes. Problematisch ist, daß der Betrachter selbst den Zusammenhang zwischen beiden Fenstern immer wieder neu erkennen muß. Hilfreich ist es hier, wenn im Übersichtsfenster markiert ist, welcher Ausschnitt auch detailliert dargestellt wird.

7.2 Visualisierungssysteme

Neben der Beschreibungen von Visualisierungstechniken finden sich in der Literatur auch Vorstellungen von implementierten dreidimensionalen Visualisierungssystemen. Die Systeme basieren auf Visualisierungstechniken, die häufig implizit mitbeschrieben werden. Diese Techniken konnten teilweise für diese Arbeit verwendet werden. Daher werden im folgenden verschiedene Systeme und die dort umgesetzten Visualisierungstechniken beschrieben.

Zunächst werden *Vogue* und *VRCS* vorgestellt. Mit *ArchView* wird danach ein System zur Darstellung von Software-Schichtenarchitekturen charakterisiert (vgl. Abschnitt 7.2.2). Schließlich wird im Abschnitt 7.2.3 *NestedVision3D* beschrieben.

7.2.1 Vogue und VRCS

Eine spezielle Form der Nutzung des dreidimensionalen Raumes liegt im Vogue-Framework vor, das von Koike vorgestellt wurde [Koi93]. Hier werden mehrere zweidimensionale Darstellungen zu einer dreidimensionalen Darstellung integriert. Ein Beispiel liefert die Darstellung von Klassenhierarchien. Sowohl die XY- als auch die YZ-Ebene werden genutzt, um jeweils ein zweidimensionales Diagramm darzustellen, konkret die jeweilige Klassenhierarchie sowie eine Liste von deklarierten Methoden. Der entstehende Zwischenraum wird genutzt, um beide Diagramme zu verbinden (vgl. Prinzipskizze in Abbildung 7.6). Als zusätzlich Information bietet die Darstellung somit gleichzeitig Aufschluß darüber, welche Klassen welche Methoden deklarieren und insbesondere wo Redefinitionen vorliegen. Die gleichzeitige Darstellung verschiedener Aspekte macht diesen Ansatz mit der dreidimensionalen Darstellung von Funktionen vergleichbar. Wie dort bekommt der Betrachter ein Gesamtbild präsentiert, daß er sich sonst erst durch mentale Integration mehrerer Bilder erarbeiten müßte.

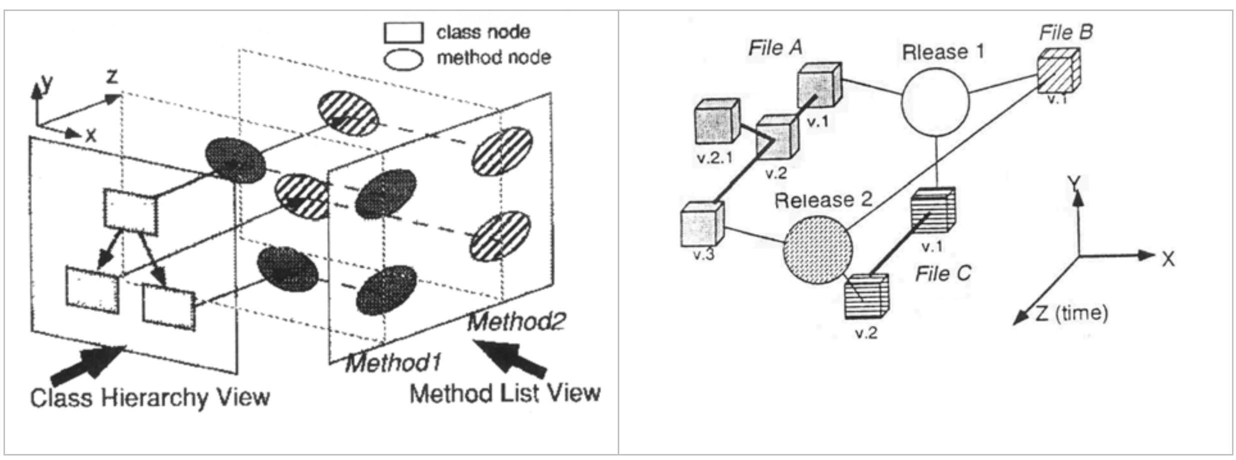


Abbildung 7.6: Prinzip der Visualisierung mit Vogue (Quelle: [Koi93])

Abbildung 7.7: Prinzip der Visualisierung mit VRCS (Quelle: [KC98])

Ein weiterer Ansatzes findet sich in dem von Koike und Chu vorgestellten Versionsmanagementwerkzeug VRCS [KC98]. Hier werden mehrere parallele Ebenen des dreidimensionalen Raumes dazu genutzt, jeweils die Versionsgeschichte einer Datei in der Form eines Baumes darzustellen (vgl. Prinzipskizze in Abbildung 7.7). Die Z-Achse repräsentiert dabei die Zeit. Zusätzlich werden in der Mitte dieser Ebenen Symbole für Versionen des Gesamtsystems dargestellt, die mit den jeweiligen Dateiversionen verbunden sind.

Koike und Chu haben ihr Versionsmanagementwerkzeug in der genannten Arbeit durch eine empirische Studie mit dem bekannten Werkzeug RCS verglichen. Dabei stellen sie fest, daß mit RCS der Zeitbedarf für eine typische Aufgabe – das Auffinden und "Auschecken" der aktuellen Version einer Datei – bis zu 4,6 mal länger dauert als mit VRCS. Sie räumen aber selbst kritisch ein, daß dies nicht allein auf die dreidimensionale Visualisierung zurück-

zuführen ist, sondern auch darauf, daß bei VRCS eine graphische Benutzungsoberfläche zur Verfügung steht, während bei RCS mit einer Kommandozeile gearbeitet werden muß. Meiner Ansicht nach ist dieser Unterschied so schwerwiegend, daß die Zeitdifferenz vor allem darauf zurückzuführen ist. Demnach würde die Studie keinerlei Rückschlüsse auf die Nützlichkeit dreidimensionaler Darstellungen erlauben.

7.2.2 ArchView

Ein weiteres Beispiel für ein dreidimensionales Visualisierungssystem ist ArchView, vorgestellt von Feijs und de Jong [FJ98]. Dabei handelt es sich um ein prototypisches System, daß vornehmlich der Visualisierung von Software mit Schichtenarchitekturen dient. Dies geschieht, indem ein Graph aus Modulen der Software und Beziehungen zwischen diesen Modulen angezeigt wird. Die Module werden dabei in äquidistante Ebenen, die parallel der XZ-Ebene verlaufen, angeordnet.

Genauer betrachtet handelt es sich bei ArchView um einen Generator für Visualisierungen. Mit verschiedenen anderen Werkzeugen werden Informationen über die darzustellende Software gewonnen. Diese dienen als Eingabe für ArchView, das daraus Quelltext in einer Beschreibungssprache für dreidimensionale Szenen erzeugt. Als Beschreibungssprache wird dabei VRML eingesetzt. Die so erzeugte Szene kann mit Hilfe eines beliebigen Anzeigeprogramm für VRML betrachtet werden.

Ein Beispiel für eine mit ArchView erzeugte Darstellung zeigt Abbildung 7.8. Hier werden verschiedenartige Module einer Software dargestellt, die über Importe miteinander verbunden sind. Die Software ist dabei in fünf Schichten eingeteilt worden.

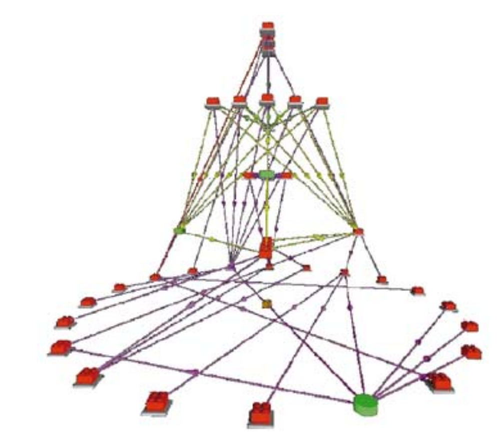


Abbildung 7.8: Beispiel einer Visualisierung mit ArchView

Bei der rechnergestützten Visualisierung von Graphen stellt sich die Frage, wie Knoten und Kanten anzuordnen sind, d.h. welche Positionen sie im Darstellungsraum einnehmen. Prinzipiell sind zwei Vorgehensweisen möglich: die Anordnung wird berechnet oder sie wird manuell – von einem Benutzer des Visualisierungssystems – angegeben. Auf dem ersten Blick hat eine automatische Erzeugung Vorteile, insbesondere im Hinblick auf den zu betreibenden Aufwand für das Erstellen einer Visualisierung. Feijs und de Jung betonen demgegenüber einen wichtigen Vorteil, den die manuelle Angabe hat: durch sie können Visualisierungen so gestaltet werden, daß sie mit der Intuition, die der Entwickler der Software über sie hatte, korrespondieren. Die manuelle Anordnung wird durch einen Editor TEDDY unterstützt, der es auch erlaubt, Knoten nach geometrischen Formen auszurichten.

Ein weiterer Punkt auf den Feijs und de Jong Wert legen, ist die Möglichkeit, Berechnungen mit Relationen durchzuführen, wobei Beziehungen zwischen Elementen des dargestellten Programmes durch Relationen modelliert werden. Beispielsweise werden sich in einem

Programm viele Module gegenseitig benutzen. Eine Darstellung aller Benutzungen könnte zu einem undurchschaubaren Gewirr von Pfeilen führen. Als eine Möglichkeit die Zahl der darzustellenden Benutzungen zu verringern, nennen Fejis und de Jong die Durchführung der Hasse-Operation auf die entsprechende Relation. Beim Vorliegen einer zyklenfreien Relation entfernt diese Operation, die auch als "transitive reduction" bezeichnet wird, Tupel aus der Relation, die – wenn die Relation als Beschreibung eines Graphen aufgefaßt wird – zu „Abkürzungen" führen. Zum Beispiel würde aus einer Relation $\{(x,y),(y,z),(x,z)\}$ das Tupel (x,z) entfernt.

7.2.3 GraphVision3D und NestedVision3D

Ein weiterer interessanter Ansatzpunkt für diese Arbeit ist das kommerzielle System *NestedVision3D* (NV3D) bzw. dessen im Forschungsumfeld entstandener Vorgänger *GraphVision3D* (GV3D), u.a. auch deshalb, weil viele Ergebnisse der Arbeit der Entwickler publiziert sind [WHF93, FW94, WF96, PFW98].

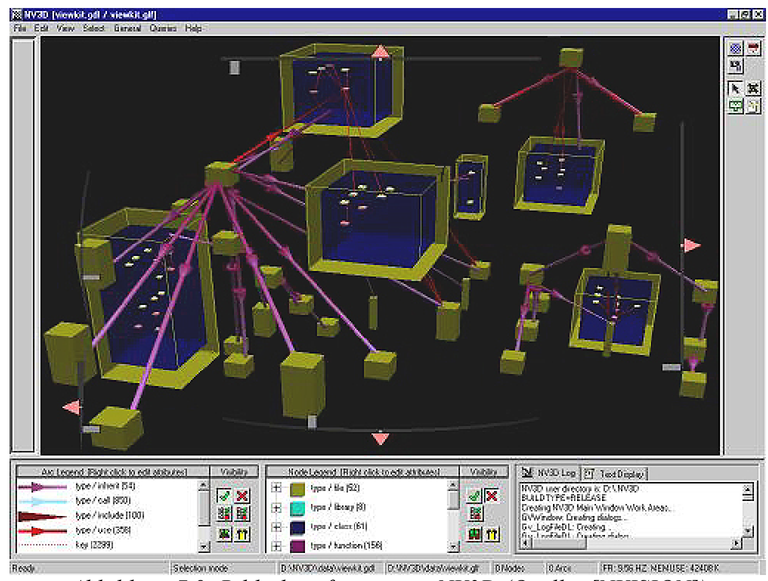


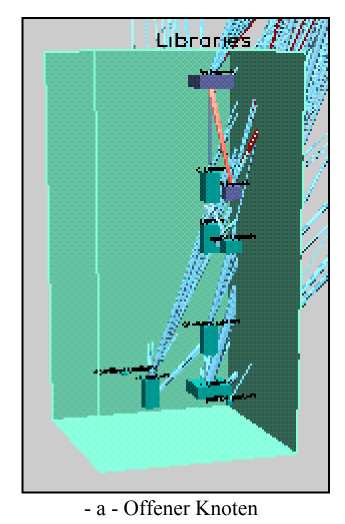
Abbildung 7.9: Bildschirmfenster von NV3D (Quelle: [NVISION])

NV3D wird zwar als allgemein für die Visualisierung komplexer Graphen geeignet beschrieben, der Schwerpunkt des Systems liegt aber bei der dreidimensionalen Visualisierung objektorientierter Software. NV3D bietet zudem Ansätze, das dynamische Verhalten von Programmen darzustellen, auf die hier nicht weiter eingegangen wird. Wie auch bei ArchView wird die Struktur der darzustellenden Software durch einen Graphen dargestellt. Bei den Knoten kann es sich um Klassen oder andere Elemente des Programmes handeln. Die Kanten werden aus Beziehungen zwischen diesen Elementen gebildet.

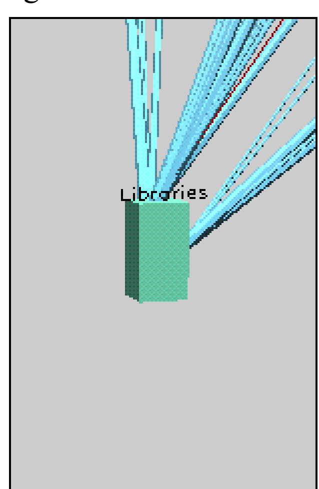
Eine Besonderheit bei GV3D und NV3D ist die Verwendung der Information-Cubes-Technik, daher auch der Name *Nested* (verschachteln). Innerhalb der Knoten können sich selbst wiederum ganze Graphen befinden, deren Knoten abermals Graphen enthalten usw. Interessant ist dies vor allem zur Darstellung von Ganzes/Teile-Beziehungen, z.B. könnten umschließende Knoten Teilsysteme der Software repräsentieren, während innere Graphen den Aufbau dieser Teilsysteme beschreiben. Dem Benutzer wird eine einfache Möglichkeit geboten, die Knoten zu öffnen, d.h. innere Graphen sichtbar zu machen, und sie wieder zu schließen. So kann er schnell zwischen einer globalen Betrachtung des Gesamtsystems und der detaillierten Untersuchung bestimmter Teilsysteme wechseln.

Wird ein Knoten geschlossen, so verringert sich auch die Anzahl der Kanten in der Darstellung, da Kanten, die von außen in Knoten des den nun nicht mehr sichtbaren Teilgraphen einlaufen, nach Möglichkeit zusammengefaßt werden (vgl. nachfolgende

Abbildung 7.10). Weitere Möglichkeiten, die Anzahl der darzustellenden Kanten zu begrenzen, bestehen darin, bestimmte Arten von Kanten – z.B. solche für Importe – nur für selektierte Knoten darzustellen oder Knoten als "tot" zu markieren. Nur Kanten lebendiger Knoten werden dargestellt. Weiterhin bietet das Programm eine als Fading bezeichnete Option, bei der alle nicht selektierten Knoten ausgegraut dargestellt werden.



- a - Offener Knoten



- b - geschlossener Knoten

Abbildung 7.10: Reduktion von Kanten durch das Schließen von Knoten

Die Entwickler von NV3D betonen, ebenso wie die von ArchView, die Vorteile des manuellen Anordnen von Darstellungen. Allerdings ist in einer neueren Version die Option enthalten, als ersten Schritt verbundene Knoten nahe beieinander plazieren zu lassen.

8 Visualisierungstechniken für Java-Software

In diesem Kapitel werden auf der Basis der im vorherigen Kapitel beschriebenen Visualisierungstechniken und -systeme Techniken zur dreidimensionalen Visualisierung des Aufbaus von Java-Software entwickelt. Damit wird die zweite konzeptionelle Frage „*Wie sollen Softwarestrukturvisualisierungen aufgebaut sein?*“ beantwortet. Diese Antwort beruht auf den während der Experimentierphase gemachten Erfahrungen des Autors.

Betrachtet man die im Abschnitt 7.1 vorgestellten Techniken Bäume, Cone Trees, Information Cubes und Information Landscapes so fällt auf, daß diese in ihrer ursprünglichen Form jeweils zur Darstellung genau einer Art von Beziehungen zwischen Elementen vorgesehen sind. Bei den ersten drei Techniken müssen die dargestellten Beziehungen zudem eine baumförmige Hierarchie bilden. In Java-Software treten jedoch verschiedene Arten von Beziehungen gleichzeitig auf. Diese Beziehungen bilden i.A. keine baumförmige Hierarchie. Dies macht deutlich, daß es nicht einfach möglich ist, eine dieser Techniken auszuwählen und damit alle Arten von Beziehungen gleichzeitig in einem Diagramm darzustellen. Eine Gleichzeitigkeit ist aber erstrebenswert, da es bei der Verwendung verschiedener Diagramme für unterschiedliche Arten von Beziehungen dem Betrachter obliegt, diese mental zu einem Gesamtbild zu verbinden. Um den Betrachter von dieser Aufgabe zu entlasten, bietet es sich an, verschiedene Arten von Beziehungen, ggf. mittels verschiedener Visualisierungstechniken, innerhalb eines Diagramms zur Ansicht zu bringen. Dies ist der Ansatz, dem im weiteren gefolgt wird. Dabei wird im Grundsatz von den semiotischen Prinzipien der körperförmigen Darstellung von Elementen und der weitgehend pfeilförmigen Darstellung von Beziehungen ausgegangen, da sich so Diagramme ergeben, die einem Betrachter auf Anhieb vertraut erscheinen.

Wie kann die geforderte Gleichzeitigkeit nun aussehen? Bevor nachfolgend auf Details eingegangen wird, gibt Abbildung 8.1 ein Beispiel. Simultan mit der Darstellung einer Klassenhierarchie durch einen Cone Tree im Vordergrund wird die Paketzugehörigkeit der einzelnen Klassen durch ein Schachtelung gemäß der Information-Cubes-Technik angezeigt. Im Hintergrund sind weitere Pakete zu sehen, so daß auch Beziehungen zwischen den Paketen in das Diagramm aufgenommen werden können. Dabei wird eine Anordnung verwendet, die an Information Landscapes erinnert, so daß sich eine Landschaft von Paketen ergibt, in der sich der Betrachter bewegen kann.

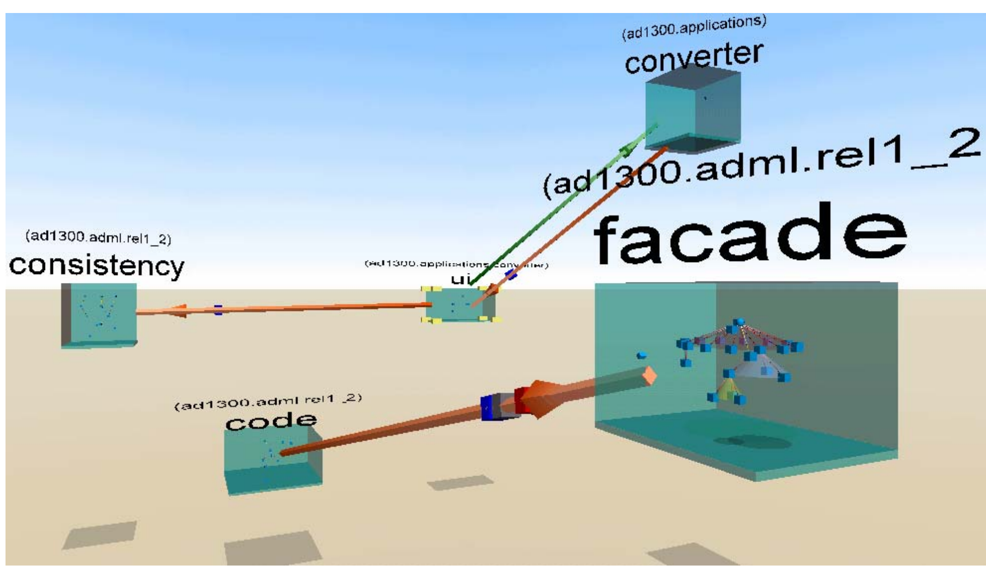


Abbildung 8.1: Beispiel für die Darstellung von Softwarestrukturen

Der Rest dieses Kapitels ist wie folgt gegliedert: In den Abschnitten 8.2 bis 8.6 werden Visualisierungstechniken für verschiedenartige Elemente und Beziehungen besprochen. Dabei wird eine Notation verwendet, die zuvor im Abschnitt 8.1 vorgestellt wurde. Im Abschnitt 8.2 wird auf die Darstellung von Entitäten und ihrer Beziehungen eingegangen. Es werden Erweiterungen zwischen Klassen (vgl. Abschnitt 8.2.1) sowie zwischen Schnittstellen (8.2.2) betrachtet, anschließend Implementierungen von Schnittstellen durch Klassen (8.2.3) und Benutzungen zwischen Entitäten (8.2.4).

Danach wird auf Pakete eingegangen (8.3), wobei die Darstellung der Paketzugehörigkeit von Entitäten (8.3.1) sowie die Präsentation von Pakethierarchien (8.3.2) zu klären ist.

Werden mehrere Pakete auf der in Abbildung 8.1 gezeigten Art dargestellt, so kann es zu u.U. langen Pfeilen zwischen den Symbolen für Entitäten verschiedener Pakete kommen. Dadurch sind Beziehungen nicht mehr leicht erkennbar. Abschnitt 8.4 behandelt dieses Problem, indem das Zusammenfassen von Paketen (8.4.1), ein temporäres Einblenden von Entitätssymbolen in „fremde" Paketsymbole (8.4.2) und eine Anzeige von Pfeilbeschreibungen (8.4.3) vorgeschlagen wird.

Die gleichzeitige Anzeige aller Gegebenheiten einer Software kann auch im Dreidimensionalen schnell zu einer Überfrachtung der Darstellung führen. Im Abschnitt 8.5 werden Maßnahmen diskutiert, um die Anzahl der dargestellten Pfeile (8.5.1) und Symbole (8.5.2) sinnvoll zu begrenzen. Zudem wird mit sogenannten Abhängigkeiten eine Möglichkeit eingeführt, Softwarestrukturen auf einem hohen Abstraktionsniveau zu betrachten (8.5.3).

Abschnitt 8.6 zeigt Möglichkeiten zur Darstellung mit der visualisierten Software assoziierter Dokumente auf.

Im Abschnitt 8.7 wird auf Probleme eingegangen, die auftreten können, wenn dem Betrachter Möglichkeiten zur Darstellungsmanipulation gegeben werden.

Abschließend erfolgt im Abschnitt 8.8 eine Zusammenfassung. Zudem werden aus den vorherigen Betrachtungen resultierende Anforderungen an ein Visualisierungssystem genannt.

Es ist zu beachten, daß die statischen zweidimensionalen Abbilder der dreidimensionalen Szenen, die in diesem Kapitel gezeigt werden, jeweils nur einen relativ schlechten Eindruck von der Szene selbst geben können. Aus diesem Grund sind einige der hier gezeigten Abbildungen auf der beiliegenden CD-ROM als dreidimensionale Szenen enthalten (vgl. Anhang A). Dort finden sich auch umfangreichere Beispiele.

8.1 Notation

In diesem Abschnitt wird eine Notation für die Darstellung von Elementen und Beziehungen vorgestellt (einen Überblick über die Notation liefert auch Anhang D). Wie bereits erwähnt, führt die dreidimensionale Darstellung zu erhöhten Anforderungen an die Gestaltung einer Notation, da Symbole und Pfeile z.B. auch aus einer größeren Entfernung der Betrachtungsposition noch erkennbar sein sollen. Als Gestaltungshilfe wurde auf die semiotischen Prinzipien nach Franck und Ware zurückgegriffen (vgl. Abschnitt 7.1.1)

8.1.1 Symbole

Abbildung 8.2 und Abbildung 8.3 auf nachfolgender Seite zeigen die für gewöhnliche Klassen und für Schnittstellen verwendeten Symbole. Als deren Vorbild diente die bekannte Unified Modeling Language (UML) [BRJ99, Oes98], in der für Klassen ein Rechteck und für Schnittstellen ein Kreis verwendet wird. Trotz ihrer einfachen Form sind die Symbole bei einer Vertrautheit mit der UML recht intuitiv. Wie auch alle weiteren Symbole sind sie so gestaltet, daß für den Betrachter ein Volumen erkennbar wird, wodurch die Assoziation zu einem greifbarem Objekt geweckt wird. Damit wird dem semiotischen Prinzip der körper-

förmigen Darstellung von Elementen entsprochen.



Abbildung 8.2: Gewöhnliche Klasse

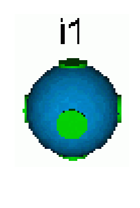


Abbildung 8.3: Schnittstelle

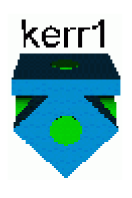


Abbildung 8.4: Fehlerklasse

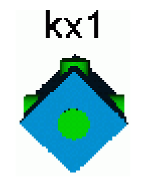


Abbildung 8.5: Ausnahmeklasse

Die Symbolfarbe – in den Abbildungen blau – kann bei der Diagrammgestaltung weitgehend frei gewählt werden. Beschriftungen befinden sich oberhalb der Symbole. Eine eventuell notwendige Ergänzung um Teile des vollständig qualifizierenden Namens, wie z.B. der Paketangabe, erfolgt oberhalb der Angabe des Entitätsnamens in einer kleineren Schriftart.

Die Beschriftung ist wegen der wechselnden Betrachtungswinkel drehbar. Bei Veränderungen des Blickwinkels auf ein Symbol, wird die Beschriftung so um die vertikale Achse des Symbols gedreht, daß sie dem Betrachter zugewandt bleibt, und nicht etwa von hinten zu sehen ist. Dies hat sich gegenüber anderen Varianten, wie z.B. einer Beschriftung auf allen Seitenwänden, als zweckmäßiger erwiesen. Neben einer verbesserten Lesbarkeit aus verschiedenen Blickwinkeln, durch welche die Objektkonstanz unterstützt wird, muß die Größe von Symbolen nicht aufgrund von u.U. langen Beschriftungen variiert werden. Eine Veränderung des Betrachtungsstandorts kann aber problematisch sein. Ist dieser weit von einem Symbol entfernt, kann die Beschriftung aufgrund der perspektivischen Verkleinerung nicht mehr entziffert werden, so daß eine Anzeige dann keinen Sinn macht.

Abbildung 8.4 und Abbildung 8.5 zeigen Symbole für weitere Arten von Klassen. Bereits in den Erläuterungen zu Java im fünften Kapitel dieser Arbeit wurden drei Arten von Klassen unterschieden, nämlich die gewöhnlichen Klassen, sowie Ausnahme- und Fehlerklassen. Diese Unterscheidung wird auch in der Notation fortgeführt. Die konzeptionelle Ähnlichkeit, die Fehler- und Ausnahmeklassen zueinander besitzen, wurde auf deren graphische Repräsentation übertragen. Dabei wurde für Fehlerklassen eine auffälligere Form gewählt, da diese als Verschärfung von Ausnahmen angesehen werden können.

In der Mitte aller sechs Seiten der gezeigten Symbole für Entitäten befindet sich die sogenannte Eigenschaftsmarkierung – in den vorangegangenen Abbildungen stets grün –, über die als relevant betrachtete Eigenschaften der Entität dargestellt werden. Dazu wird die Markierung in den Dimensionen Form und Farbe variiert. Dies geschieht unabhängig voneinander, so daß dem semiotischen Prinzip der Orthogonalität nachgekommen wird. Die Wiederholung auf allen sechs Seiten verbessert die Erkennbarkeit.

Über die Farbe der Eigenschaftsmarkierung wird der Zugriffsmodus einer Entität angegeben. Es gelten die in Tabelle 8.1 auf nachfolgender Seite angegebenen Entsprechungen zwischen den Zugriffsmodi und der Farbe der Eigenschaftsmarkierung. Diese Korrespondenzen sind an die Farben einer Ampel angelehnt, wobei zusätzlich grau für den Modus *private* verwendet wird, was dazu führen soll, daß für das globale Verständnis potentiell weniger wichtige Entitäten optisch etwas zurücktreten. Die Vergabe der Ampelfarben erfolgt aus einer Sicht von außerhalb des Paketes zu dem eine Entität gehört. So ist ein externer Zugriff auf Entitäten mit dem Zugriffsmodus *default* gesperrt, weswegen hier rot verwendet wird.

Die Instanziierbarkeit einer Entität wird durch die Form der Markierung angezeigt (vgl. Tabelle 8.2 auf nachfolgender Seite). Kreis und Dreieck sind bei mittleren Entfernungen noch recht gut unterscheidbar, zudem korrespondiert der Kreis zu den Kugeln für Schnittstellen. Da Schnittstellen stets abstrakt sind, kann der Kreis wegen der Korrespondenz gut mit der Bedeutung *abstrakt* in Verbindung gebracht werden. Der Balken am unteren Ende des

Dreiecks bei *final* kann leicht mit einer Sperrung – genauer: der Sperrung weiterer Spezialisierungen – assoziiert werden.

Zugriffsmodus	Farbe	Beispiel
public	Grün	●
protected	Orange	●
default	Rot	●
private	Grau	●

Tabelle 8.1: Farbkodierung der Zugriffsmodi

Instanziierbarkeit	Form	Beispiel
abstrakt	Kreis	●
konkret	Dreieck	▼
final	Dioden-zeichen	▼

Tabelle 8.2: Formkodierung der Instanziierbarkeit

Ausführbare Klassen sind Einstiegspunkte in die Software und erhalten daher eine gesonderte Kennzeichnung über eine weitere Variation der Form der Eigenschaftsmarkierung. Bei Symbolen für ausführbare Klassen wird die Eigenschaftsmarkierung durch zwei kleine Rechtecke umschlossen (vgl. Abbildung 8.6).

Ebenso wie bei Beschriftungen gilt auch für Eigenschaftsmarkierungen, daß sie aus großen Entfernungen nicht mehr erkennbar sind. Da sie relativ detaillierte Informationen über Elemente vermitteln, ist dies nicht problematisch, eine Anzeige kann daher für weit entfernte Symbole unterbleiben.

Autovermietung

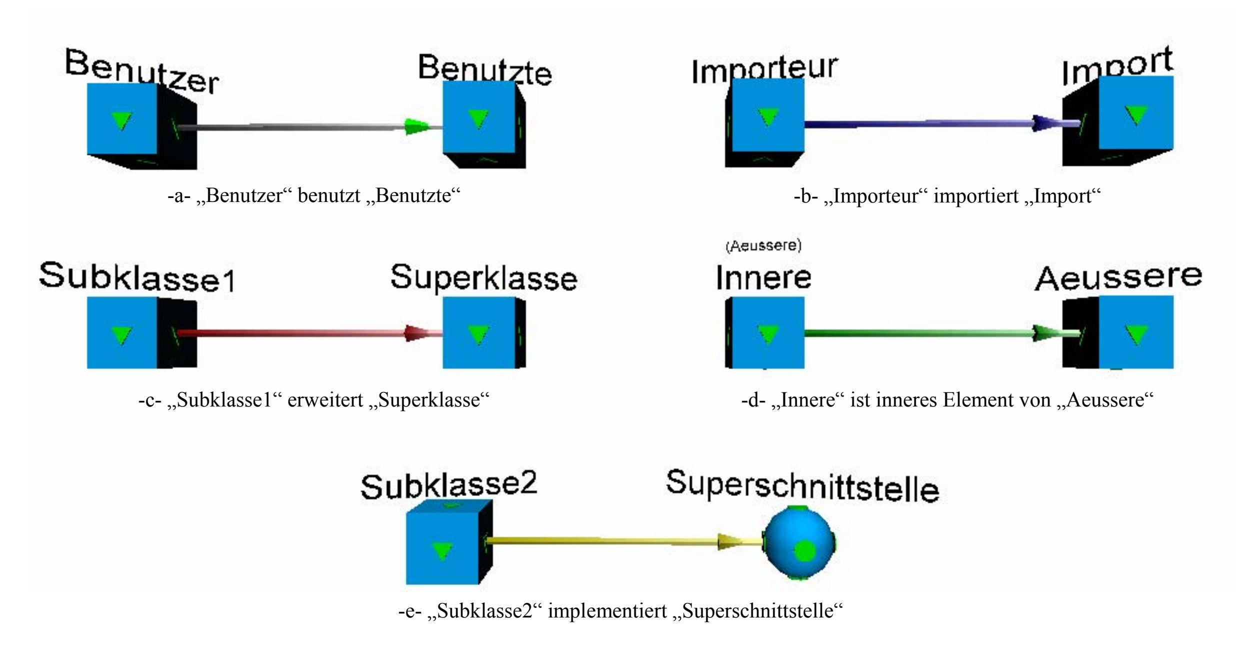
Abbildung 8.6: Eigenschaftsmarkierung bei ausführbaren Klassen

Schließlich könnte man sich noch vorstellen, über die Größe von Symbolen quantitative Eigenschaften von Elementen zu verdeutlichen, so beispielsweise den unterschiedliche Umfang des Quelltextes der Entitäten. Es hat sich aber während dieser Arbeit gezeigt, daß bei dreidimensionalen Darstellungen die Größe von graphischen Objekten kaum nutzbar ist, um Informationen zu vermitteln (vgl. hierzu auch [RS99, S.93]). Dies gilt zumindest dann, wenn keine spezielle Hardware für die Unterstützung des Tiefeneindrucks zur Verfügung steht, da die Größe von Objekten in solchen Fällen stärker mit der perspektivischen Verzerrung in Verbindung gebracht wird. Objekte erscheinen eher unterschiedlich tief in der Darstellung als unterschiedlich groß. Simulierte Schatten können bei wechselnden Betrachtungspositionen und Blickrichtungen nur unzureichende Abhilfe schaffen, da sie leicht aus dem Blickfeld geraten.

8.1.2 Pfeile

Die Pfeile für verschiedene Arten von Beziehungen werden durch ihre Färbung voneinander unterschieden (vgl. Abbildung 8.7 auf nachfolgender Seite). Dies hat sich wegen einer relativ guten Unterscheidbarkeit als zweckmäßig erwiesen. Die Richtung von Pfeilen wird wie gewöhnlich durch ihre Spitzen angegeben. Bei einer dreidimensionalen Darstellung verstärkt sich aber das Problem, diese geeignet zu dimensionieren. Zu große Spitzen können störend wirken, während bei kleiner Spitzen bereits aus relativ geringer Entfernung die Richtung nicht mehr leicht erkennbar ist. Franck und Ware schlagen deshalb eine Verwendung von Helligkeitsverläufen zum kenntlich machen der Richtung vor [FW94]. Eine derartige Verwendung ist auch in Abbildung 8.7 erkennbar.

Im Zusammenhang mit der Simulation einer Beleuchtung können sich bei Helligkeitsverläufen aber störende optische Effekte ergeben. In Einzelfällen kann sich eine Verwirrung darüber einstellen, ob eine Aufhellung durch die Beleuchtung oder im Zuge der Richtungsangabe entstanden ist, so daß eine Verwendung von Pfeilspitzen weiter notwendig bleibt.



-a- „Benutzer“ benutzt „Benutzte“

-b- „Importeur“ importiert „Import“

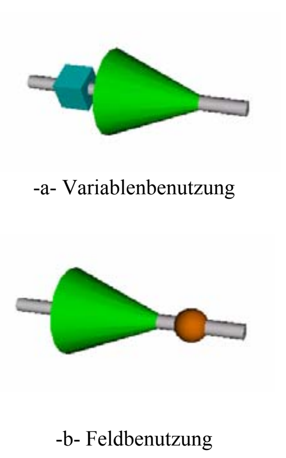
-c- „Subklasse1“ erweitert „Superklasse“

-d- „Innere“ ist inneres Element von „Aeussere“

-e- „Subklasse2“ implementiert „Superschnittstelle“

Abbildung 8.7: Verwendete Arten von Pfeilen und ihre Bedeutung

Viele bestehende Notationen nutzen die Abwandlung von Pfeilspitzen zur Vermittlung weiterer Informationen. Dies ist auch hier möglich, da die so zu vermittelnden Details nicht unbedingt aus größerer Entfernung erkennbar sein müssen. Bei Benutzungsbeziehungen kann beispielsweise angezeigt werden, welche Formen der Benutzung vorliegt. In Anlehnung an UML werden hier die Deklaration von Variablen und die Benutzung von Feldern gesondert hervorgehoben. Abbildung 8.8 zeigt die beiden entsprechenden Erweiterungen. Die Formen der Erweiterungen sind ebenfalls an die UML angelehnt, sie werden aber etwas vereinfacht. In der UML wird eine Aggregation durch ein rautenförmiges Symbol angezeigt, welches mit dem für die Verwendung von Variablen benutzten Quader vergleichbar ist. Die Multiplizitätsangabe erfolgt für eine Kardinalität *n* durch einen Stern, der hier durch eine Kugel angenähert wird. Liegen beide Sonderformen der Benutzung vor, so werden die entsprechenden Erweiterungen kombiniert.



-a- Variablenbenutzung

-b- Feldbenutzung

Abbildung 8.8: Anzeige der Details einer Benutzung

Der Zugriffsmodus wird mit Hilfe von Färbung der Pfeilspitze bzw. deren Erweiterungen angezeigt. Dabei wird die bereits in Tabelle 8.1 auf Seite 50 angegebenene Farbkodierung verwendet. Die Farbe der Pfeilspitze wird durch den höchsten vorliegenden Zugriffsmodus bestimmt. Abbildung 8.9 gibt ein Beispiel und zeigt auch die Zusammenfassung von Pfeilen bei symmetrischen Beziehungen, die zur Reduktion der Gesamtzahl der benötigten Pfeilen eingesetzt werden kann. Zur Verdeutlichung ist neben der Abbildung ein entsprechender Quelltextausschnitt aufgeführt.



Abbildung 8.9: Beispiel einer symmetrischen Benutzung

```
class E1 {
    private E2   alsVar;
    public  E2[] alsFeld() { ... }
}
class E2 {
    public  E1   einfach() { ... }
}
```

Es kann vorkommen, daß zwischen zwei Elementen gleichzeitig mehr als eine Beziehung

besteht. In diesen Fällen besteht die Aufgabe, die verschiedenen Pfeile optisch voneinander zu trennen. Dies erfolgt hier durch eine Variation der Ansatzpunkte der Pfeile an den Symbolen (vgl. Abbildung 8.10).



Abbildung 8.10: Gleichzeitige Erweiterungen mit Redefinition und Benutzung von "E4" durch "E3"

Abbildung 8.10 zeigt zudem, wie durch eine doppelte Pfeilspitze Erweiterungen mit Redefinitionen hervorgehoben werden können. Dieses Detail der Notation wurde allerdings im Rahmen dieser Arbeit nicht implementiert, so daß in den nachfolgenden Abbildungen keine Hervorhebungen gezeigt werden.

8.2 Entitäten und ihre Beziehungen

8.2.1 Erweiterungen zwischen Klassen

Die Erweiterungen zwischen Klassen ist bei objektorientierten Programmen von besonderer Relevanz, die sich auch darin zeigt, daß viele Methoden zum objektorientierten Softwareentwurf wie bspw. der Booch-Methode [Boo95] den Entwickler dazu anhalten, Vererbungshierarchien zwischen Klassen zu bilden. Aus diesem Grund soll mit der Entwicklung einer Darstellungstechnik für Klassenhierarchien begonnen werden.

In Java handelt es sich durch die fehlende Unterstützung der Mehrfacherbung um baumförmige Vererbungshierarchien. Dadurch wird die Verwendung der im Kapitel 7 beschriebenen Techniken zur Darstellung baumförmiger Hierarchien möglich. Betrachtet man sich die Gegebenheiten genauer, so stellt man fest, daß Java-Software immer genau eine Vererbungshierarchie besitzt, da alle Klassen von der gemeinsamen Oberklasse *Object* erben. Es wäre somit möglich, alle Klassen z.B. innerhalb eines Cone Trees darzustellen. Einige Klassen der Java-API, insbesondere solche zur Verwaltung von Kollektionen von Objekten, basieren zwar auf der Existenz der gemeinsamen Oberklassen, bei der Modellierung von Programmen wird sie aber nur selten von praktischer Bedeutung sein. Es ist daher häufig eher sinnvoll, die Hierarchie in mehrere Teilhierarchien aufzuspalten.

Zunächst bietet sich die Verwendung der Cone-Tree-Technik an. Abbildung 8.11 auf nachfolgender Seite zeigt eine Klassenhierarchie aus dem Paket *awt* der Java-API. Insbesondere für größere Hierarchien ist die Technik geeignet. So konnten mit ihr eine Klassenhierarchie bestehend aus allen Klassen der Pakete *java.awt* und *java.lang* visualisiert werden (vgl. Anhang A). Diese Hierarchie umfaßt 137 Klassen und hat damit eine Größe, die sich auf herkömmliche Weise kaum übersichtlich darstellen läßt. Dabei besitzen Cone Trees im Aussehen gewisse Ähnlichkeit zu der bei zweidimensionalen Darstellungen häufig verwendeten Darstellung durch Bäume, so daß damit gerechnet werden kann, daß Betrachter, die mit der Darstellung durch Bäume vertraut sind, sich schnell an die Cone-Tree-Technik gewöhnen.

Weit weniger intuitiv ist die Verwendung von Information Cubes zur Darstellung von Vererbungshierarchien. Da diese aber auch für mehr als 1000 Knoten, d.h. hier Klassen, geeignet sein sollen, könnte ihr Einsatz zwar bei Hierarchien angebracht sein, die selbst für Cone Trees zu groß sind, normalerweise dürfte eine Aufspaltung der Hierarchie in mehrere Teilhierarchien hier aber günstiger sein.

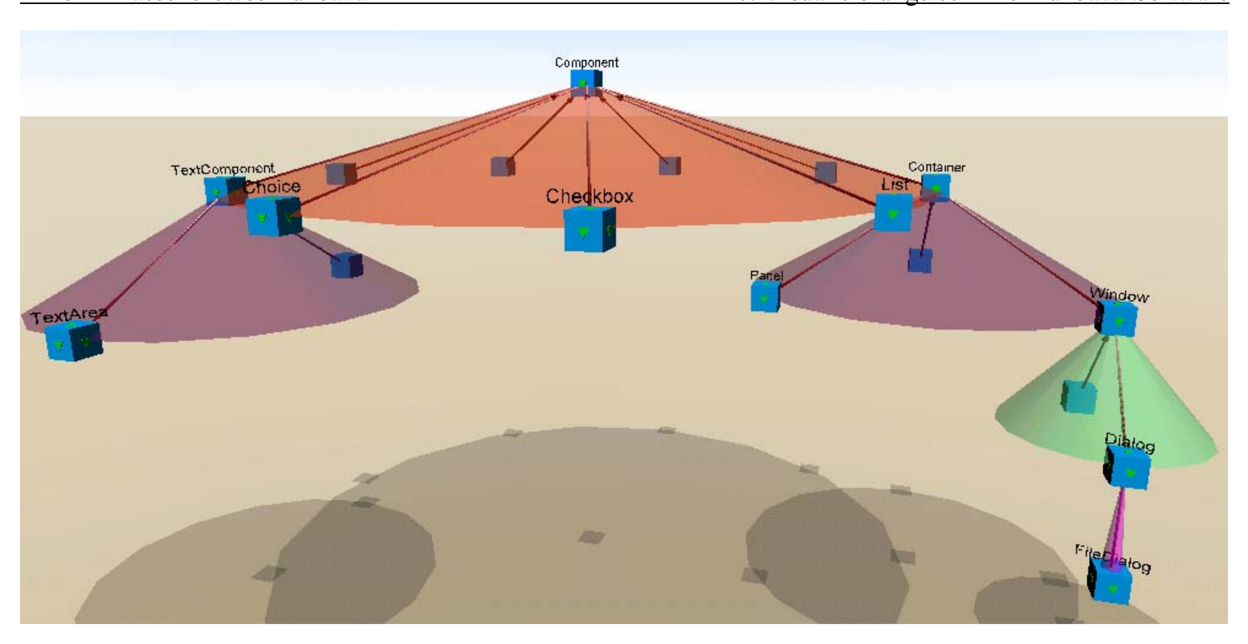


Abbildung 8.11: Darstellung einer Klassenhierarchie als Cone Tree

Auch die herkömmliche Darstellung einer Vererbungshierarchie als Baum kann im dreidimensionalem Raum eingesetzt werden. Gegenüber einer zweidimensionalen Darstellung ergibt sich der Vorteil, daß der jeweilige Baum aus verschiedenen Blickwinkeln betrachtet werden kann. Hierbei wird besonders deutlich, wie die perspektivische Verzerrung dazu führt, daß neben einigen fokussierten Klassen im Vordergrund viele weitere verkleinert im Hintergrund sichtbar sind und so der Kontext erkennbar bleibt (vgl. Abbildung 8.12).

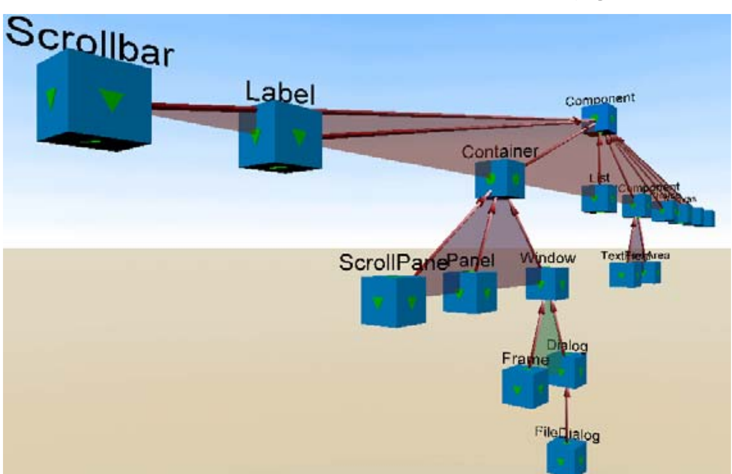


Abbildung 8.12: Schräger Blick auf eine Klassenhierarchie als Baum

Durch den Wunsch, mehrere Aspekte in einem Diagramm darzustellen, also bspw. mit den Erweiterungen auch Benutzungen zwischen Klassen zu zeigen, kommt es dazu, daß die Pfeile für Erweiterungen nicht die einzigen sind, welche die Klassen in einem Cone Tree, einem Information Cube oder einen Baum berühren. Insbesondere die Cone-Tree-Darstellung führt dabei zu Problemen, da der Innenraum von Cone Trees bereits so stark genutzt ist, daß es schnell zu Überschneidungen kommt, wenn bspw. viele Benutzungsbeziehungen zwischen den enthaltenen Klassen bestehen. Verstärkt wird dieses Problem noch durch die in der Cone-Tree-Technik integrierte interaktive Drehung der einzelnen Kegel, da durch die Drehung immer neue Anordnungen der Symbole entstehen, welche den Verlauf der weiteren Pfeile nicht berücksichtigen. Auch die Darstellung als Baum ist diesbezüglich problematisch. Beziehungen zu Klassen außerhalb des Baumes können relativ gut gezeigt werden. Existieren aber viele andersartige Beziehungen zwischen den Klassen der Erweiterungshierarchie,

kommt es häufig zu Überschneidungen. Für solche Situationen wird die nachfolgend vorgestellte Visualisierungstechnik vorgeschlagen – die kegelförmige Darstellung (vgl. Abbildung 8.13). Hier werden alle Symbole auf der Mantelseite eines Kegels ähnlich zur Baumdarstellung angeordnet. Der Innenraum bleibt frei um andersartige Beziehungen anzuzeigen. Weiterhin kann er auch genutzt werden, um Entitäten aufzunehmen, die zur Hierarchie vergleichsweise vielen Beziehungen haben (vgl. Abbildung 8.14). Existieren solche Beziehungen oder Entitäten aber nicht, so bleibt der Innenraum des Kegels ungenutzt. Hier ist die kegelförmige Darstellung somit weniger geeignet.

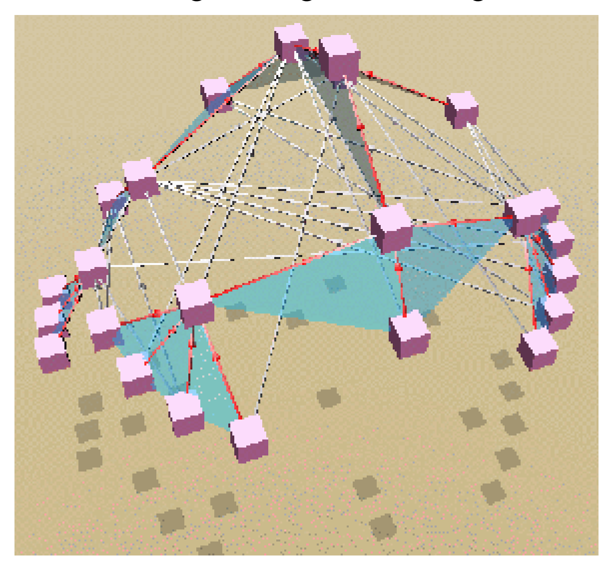


Abbildung 8.13: Kegelförmige Darstellung bei vielen Beziehungen zwischen den Klassen der Hierarchie

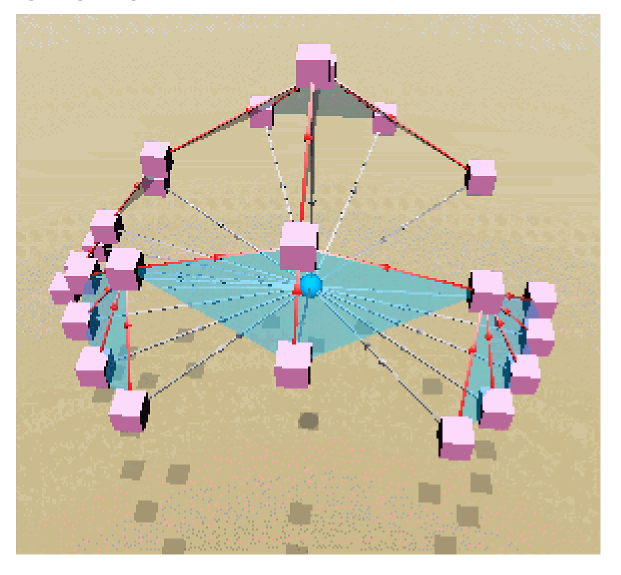


Abbildung 8.14: Kegelförmige Darstellung mit Nutzung des Innenraum für hierarchiefremde Entität

Sowohl Cone Trees als auch Kegel und Bäume geben die Anordnung der Symbole weitgehend vor. Wie oben gezeigt wurde, können durch sie Erweiterungshierarchien gut dargestellt werden. Die gleichzeitige Darstellung andersartiger Beziehungen ist aber problematisch. Kegel können hier in einigen Fällen Abhilfe schaffen. Sie sind aber durch die mangelnde Ausnutzung des Innenraumes nicht in allen Fällen günstig. Als weitere Alternative bietet sich an, die Vorgaben für die Anordnung zu lockern, so daß ebenenförmige oder auch top-down-Darstellungen entstehen. Die Lockerungen führen zu einer besseren Darstellbarkeit von zusätzlichen Beziehungen, da diese bei der Plazierung von Symbolen leichter berücksichtigt werden können, die Erweiterungshierarchie tritt aber weniger deutlich hervor. Ein Beispiel dafür liefert Abbildung 8.17 auf der Seite 56, in der eine Klassenhierarchie kombiniert mit Implementierungsbeziehungen gezeigt wird.

Der Einsatz ebenenförmiger Darstellungen zeigt einen weiteren Vorteil dreidimensionaler Visualisierungen auf – gegebene Visualisierungstechniken lassen sich hier flexibler umsetzen. So führt die Ebenenförmigkeit im Zweidimensionalen zu dem Zwang, Symbole einer Ebene auf einer horizontalen Linie zu plazieren. Dreidimensionale Darstellungen bieten mehr Gestaltungsspielraum, da zur Positionierung der Symbole einer Ebene eine Fläche zur Verfügung steht. Somit können Symbole unter Beibehaltung der Ebenenförmigkeit nicht nur nebeneinander sondern auch hintereinander angeordnet werden, wodurch lange Pfeile und Überschneidungen vermindert werden können. Vergleichbares gilt auch für Bäume, die mit verschiedenen Drehwinkeln zur Y-Achse verwendet werden können, wobei die Eigenschaft, daß Superklassen oberhalb von Subklassen plaziert sind, unabhängig davon erhalten bleibt. In zweidimensionalen Darstellungen ist mit einer Variation der Ausrichtung eines Baumes der Verlust dieser Eigenschaft verbunden, was insbesondere dann das Verständnis erschweren kann, wenn verschieden ausgerichtete Bäume gleichzeitig verwendet werden.

8.2.2 Erweiterungen zwischen Schnittstellen

Im Gegensatz zu Erweiterungshierarchien von Klassen sind aus Schnittstellen bestehende Erweiterungshierarchien nicht immer baumförmig. Liegt aber eine Baumförmigkeit vor, so können die im letzten Abschnitt aufgeführten Überlegungen übertragen werden. Bei den betrachteten Softwarebeispielen fiel allerdings auf, daß umfangreiche Erweiterungshierarchien bei Schnittstellen weit weniger zu erwarten sind als bei Klassen. Somit ist der Einsatz spezieller Techniken hier i.d.R. wenig sinnvoll und es kann für nicht baumförmige Hierarchien gut auf die ebenenförmige oder die top-down-Darstellung zurückgegriffen werden.

8.2.3 Implementieren von Schnittstellen durch Klassen

Die Verwendung einer dreidimensionalen Darstellung bieten den Vorzug, daß mehrere Hierarchien hintereinander dargestellt werden können. Benutzt man hintereinander liegende Flächen abwechselnd für Klassen und Schnittstellen, entsteht eine Darstellung der Implementierungsbeziehung, die den Abbildungen ähnelt, wie sie durch das im Abschnitt 7.2.1 beschriebene Versionsmanagementwerkzeug VRCS (vgl. Abbildung 8.15) erzeugt werden. Auf diese Art können Zusammenhänge zwischen den Hierarchien gut verdeutlicht werden, wobei man sich nicht auf Implementierungen beschränken muß, sondern auch z.B. Benutzungen einzeichnen kann.

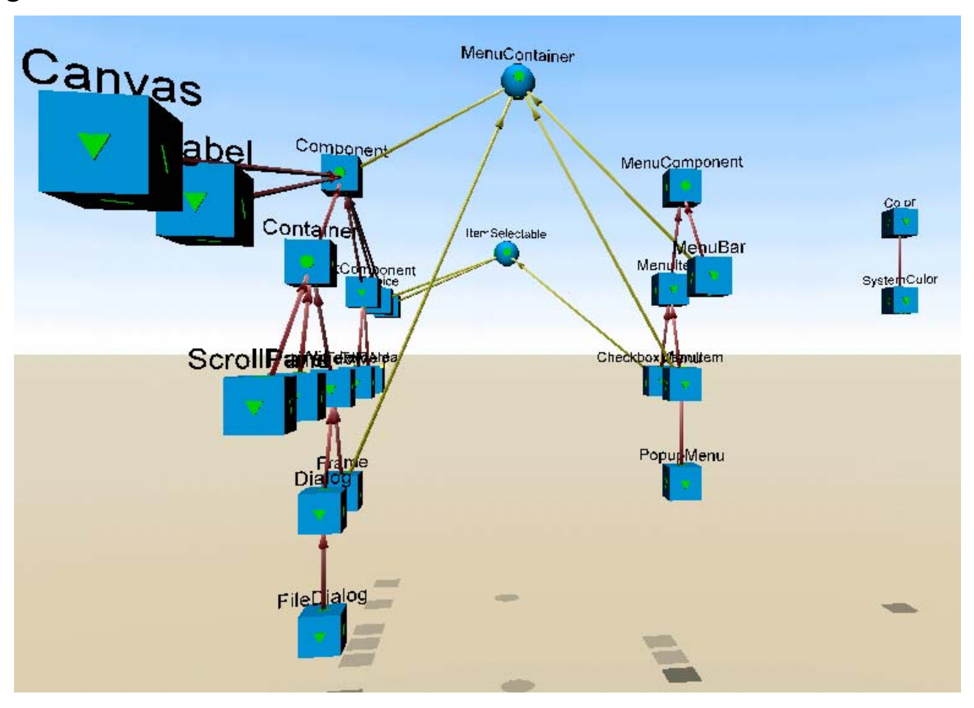


Abbildung 8.15: Darstellung mehrerer Hierarchien in Anlehnung an VRCS

Erweiterungshierarchien zwischen Schnittstellen und Klassen kann man, da sie sich auf verschiedene Arten von Entitäten beziehen, als orthogonal bezeichnen – im Sinne von unabhängig voneinander. Im dreidimensionalen Raum ergibt sich die Möglichkeit dies durch eine geometrische Orthogonalität zu versinnbildlichen (vgl. Abbildung 8.16 auf nachfolgender Seite).

Aufgrund der verhältnismäßig kleinen zu erwartenden Schnittstellenhierarchien und auch der weit geringeren Zahl von zu erwartenden Schnittstellen gegenüber Klassen insgesamt - beispielsweise stehen den ca. 600 Klassen der Java-API lediglich ca. 135 Schnittstellen gegenüber – wird es oft auch ausreichend sein, lediglich eine ebenenförmige- oder top-down-Darstellung zu benutzen, wobei die implementierten Schnittstellen oberhalb der

implementierenden Klassen angeordnet werden. Somit ergibt sich ein Bild, bei dem durchgängige Superentitäten oberhalb von Subentitäten angeordnet sind (vgl. Abbildung 8.17). Diese Einheitlichkeit trägt zum schnelleren Erfassen der Darstellung bei.

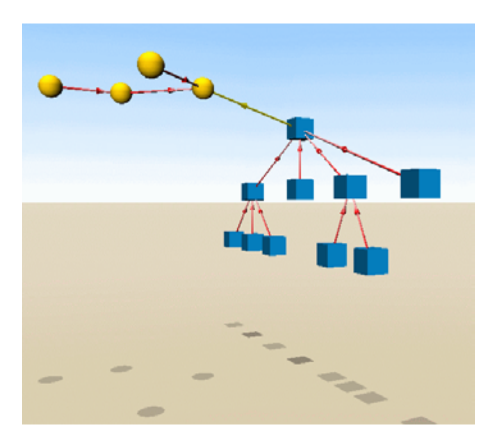


Abbildung 8.16: Orthogonale Darstellung der Implementierung

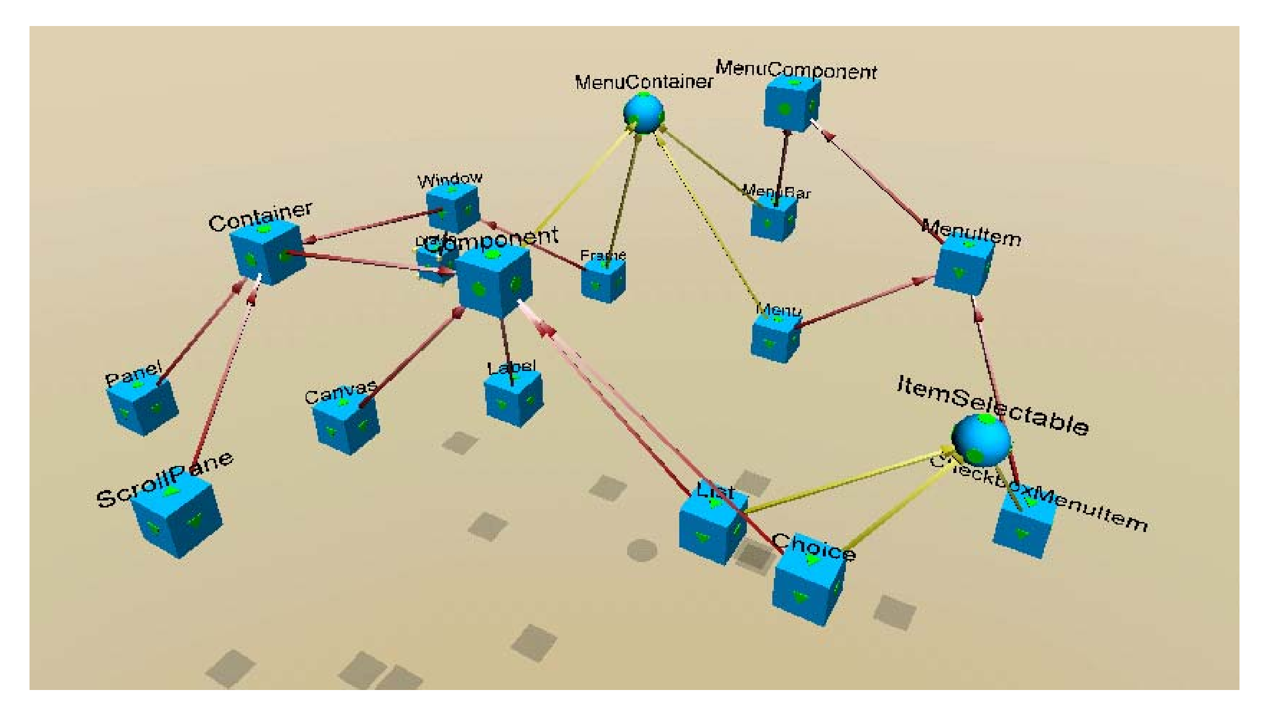


Abbildung 8.17: Top-Down-Darstellung von Implementierungen zusammen mit Erweiterungen

8.2.4 Benutzungen

Im Gegensatz zu den zuvor betrachteten Beziehungen können Benutzungen zyklisch sein, so daß keine Hierarchie vorliegt. Damit ist die ebenenförmige bzw. die top-down-Darstellung nicht vollständig umsetzbar. Trotzdem ist es erstrebenswert, zumindest die Anzahl der gegen das top-down-Prinzip verstoßenden Pfeile zu minimieren, um den Aufbau der Darstellung strukturiert zu halten. Während es bei den verschiedenen Hierarchien allerdings leicht ersichtlich war, welches Symbol oberhalb eines anderen Symbols anzuordnen ist, nämlich das der Superentität über dem der Subentität, muß dies bei der Benutzung genauer untersucht werden. Hier kann argumentiert werden, daß benutzende Entitäten abstrakter ein Problem beschreiben, während benutzte Entitäten konkreter zu dessen Lösung verwendet werden. Analog zu Erweiterungshierarchien würden Symbole für benutzende, also abstraktere, Entitäten oberhalb von Symbolen für konkreteren Entitäten plaziert. Dabei kommt es allerdings zu einer Ausrichtung von Pfeilen, die der bisher vorherrschenden Pfeilrichtung –

von unten nach oben – entgegen läuft. Dies scheint aber akzeptabel, da die unterschiedlich ausgerichteten Pfeile unterschiedliche Bedeutungen haben.

Die nach der Benutzungsbeziehung ermittelte vertikale Anordnung zwischen Entitäten kann mit der sich aus den Erweiterungen und Implementierungen ergebenden Anordnung in Konflikt stehen. Gemäß einer schwächeren strukturellen Bedeutung der Benutzung, die i.d.R. vorausgesetzt werden kann, sollte bei gleichzeitiger Anzeige normalerweise die aus der Erweiterung bzw. Implementierung abgeleitete Anordnung den Ausschlag geben.

8.3 Pakete

8.3.1 Paketzugehörigkeit

Die Paketzugehörigkeit von Entitäten bietet einen sinnvollen Ansatzpunkt zur Aufteilung der Darstellung in mehrere Bereiche, die getrennt betrachtet werden können und damit leichter verständlich sind. Die Visualisierung der Paketzugehörigkeit kann bei einer dreidimensionalen Darstellung in Anlehnung an die Information-Cubes-Technik geschachtelt erfolgen, wobei Entitätssymbole in das entsprechende Paketsymbol eingebettet werden (vgl. Abbildung 8.18).

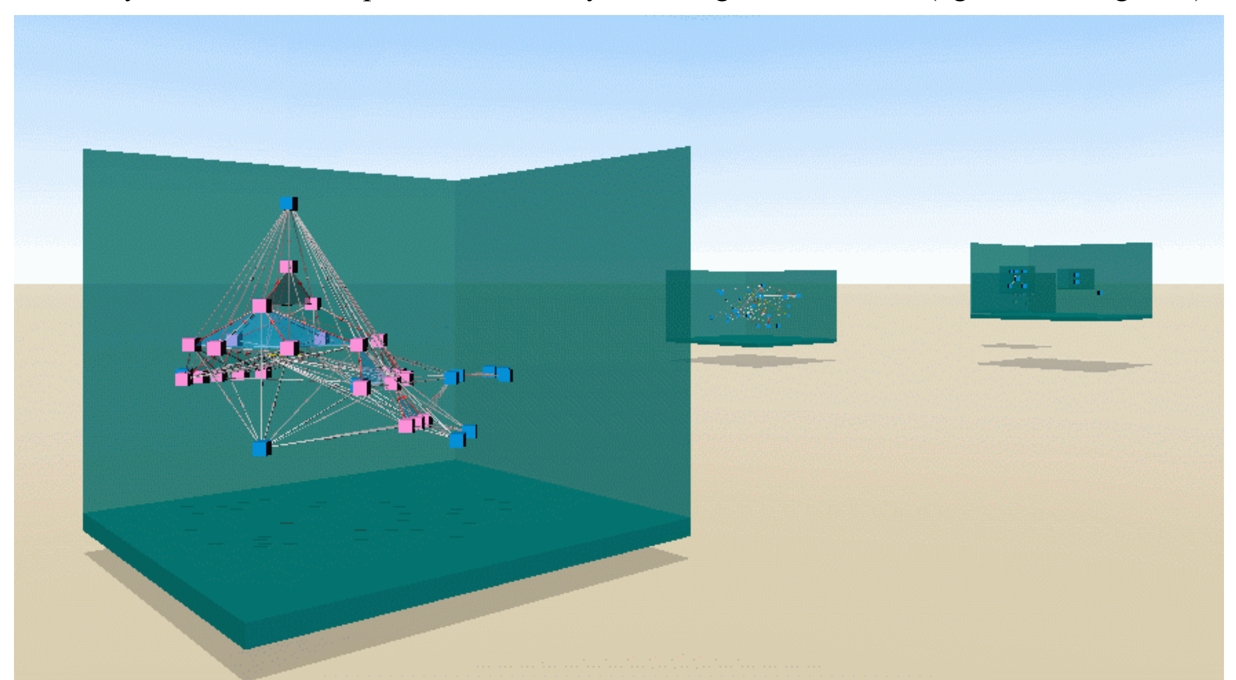


Abbildung 8.18: Darstellung der Paketzugehörigkeit mit Hilfe von Schachtelung

Schachtelungen von Symbolen sind auch bei zweidimensionalen Darstellungen möglich. Bei einer dreidimensionalen Darstellung ergibt sich aber neben der bereits einleitend erwähnten größeren Deutlichkeit der Vorteil, daß der Betrachter seinen Standort wechseln und in das Paket „eintauchen“ kann, um sich so in diesem umzusehen und sich über Details zu informieren. Er kann nahtlos von einer Darstellung, die einen Überblick über mehrerer Pakete liefert, zu den Details eines Pakets wechseln. Diese Nahtlosigkeit des Übergangs fehlt bei herkömmlichen zweidimensionalen Visualisierungssystemen häufig, bei denen es, wenn eine Umsetzung der Rapid Zooming Technik fehlt, nur möglich ist, aus vorgegebenen Skalierungsfaktoren auszuwählen. Die Veränderung eines Skalierungsfaktors kann zu einer sprunghaften Veränderung der Anzeige führen, nach der sich der Betrachter erst erneut orientieren muß.

Wird eine zweidimensionale Darstellung stark vergrößert, so daß auch Details von Symbolen tieferer Schachtelungsebenen sichtbar werden, kann die Gesamtübersicht schnell verloren

gehen. Die vorangegangene Abbildung 8.18 macht deutlich, wie die perspektivische Verzerrung genutzt werden kann, um neben dem vergrößertem Paket im Vordergrund gleichzeitig eine Übersicht über weitere Pakete zu zeigen.

Die Wände der verwendeten Paketsymbole erinnern an halb durchsichtige Spiegel. Von außen kann ungehindert hineingeguckt werden, die Wand wird maximal angedeutet bzw. entfällt ganz. Demgegenüber kann von innen nur eingeschränkt herausgesehen werden. Dadurch treten bei der Betrachtung des Paketes außerhalb liegende graphische Objekte in den Hintergrund. Durch eine leichte Durchsichtbarkeit bleibt die Einbettung des Paketes in den Gesamtzusammenhang aber sichtbar. Geschachtelte Symbole können flexibel angeordnet werden. So bleibt die Verwendung spezieller Visualisierungstechniken z.B. zur Darstellung von Klassenhierarchien möglich.

Wie Abbildung 8.18 ebenfalls zeigt, werden Paketsymbole in einer an die Information Landscapes erinnernden Form angeordnet, so daß sich eine Landschaft von Paketen ergibt, in der sich der Betrachter bewegt. Gegenüber dem ursprünglichen Konzept der Information Landscapes müssen die Symbole aber nicht auf dem Boden der Darstellung fußen, sondern können verschiedene Höhen besitzen. Dadurch verliert das Bild zwar etwas von seiner Natürlichkeit, Überschneidungen von Pfeilen können aber besser vermieden werden. Simulierte Schatten dienen dazu, die Höhe zu verdeutlichen und den Tiefeneindruck zu verstärken.

8.3.2 Pakethierarchie

Pakethierarchien sind in Java baumförmig. Im Allgemeinen existieren, z.B. durch die Verwendung von Klassenbibliotheken, mehrere Hierarchien in einer Software. Für jede einzelne Hierarchie stehen aufgrund der Baumförmigkeit die im Abschnitt 8.2.1 genannten Visualisierungstechniken für Klassenhierarchien zur Verfügung. Allerdings muß deren Bewertung hier anders ausfallen. So ist die Verwendung einer Schachtelung von Symbolen, d.h. der Information-Cubes-Technik, an dieser Stelle eingängiger als z.B. für Vererbungshierarchien. Sie ist insbesondere dann angebracht, wenn Subpakete als Details ihres Superpaketes gesehen werden sollen, die in der Gesamtsicht weitgehend unwichtig und deshalb zu verdecken sind. Bei der Schachtelung werden erst bei näherer Betrachtung eines Paketes dessen Subpakete deutlich sichtbar. Will man jedoch die Pakethierarchien als Ganzes übersichtlich darstellen, ist die Verwendung von Cone Trees oder dergleichen besser geeignet, da zur Betrachtung tief in der Hierarchie liegender Subpakete bei der Information-Cubes-Technik erst ein Eintauchen in eine ganze Reihe von Symbolen erforderlich ist.

8.4 Beziehungen zwischen Entitäten verschiedener Pakete

Zwischen Entitäten verschiedener Pakete können alle im Abschnitt 8.2 erwähnten Formen von Beziehungen bestehen, zudem sind hier Importe zu erwarten[1]. Durch die Einteilung des dreidimensionalen Raumes in ausschließlich für ein Paket genutzte Bereiche kann es dazu kommen, daß zur Anzeige paketübergreifender Beziehungen verhältnismäßig lange Pfeile notwendig werden, die einer leichten Erkennbarkeit sehr entgegen stehen. In einer Ansicht, die einen Überblick über die vorhandenen Pakete bieten soll, kann eine Anzeige derartiger Beziehungen trotzdem von Vorteil sein, da sie eine Tendenz erkennen läßt, welche und wie viele Beziehungen zwischen den Paketen bestehen (vgl. Abbildung 8.19 auf nachfolgender Seite).

[1] Importe zwischen Entitäten eines Pakets werden durch die Java-Sprachdefinition nicht verboten, sie sind aber nicht sinnvoll, da auch ohne Import einer Entität alle anderen Entitäten desselben Pakets zur Verfügung stehen.

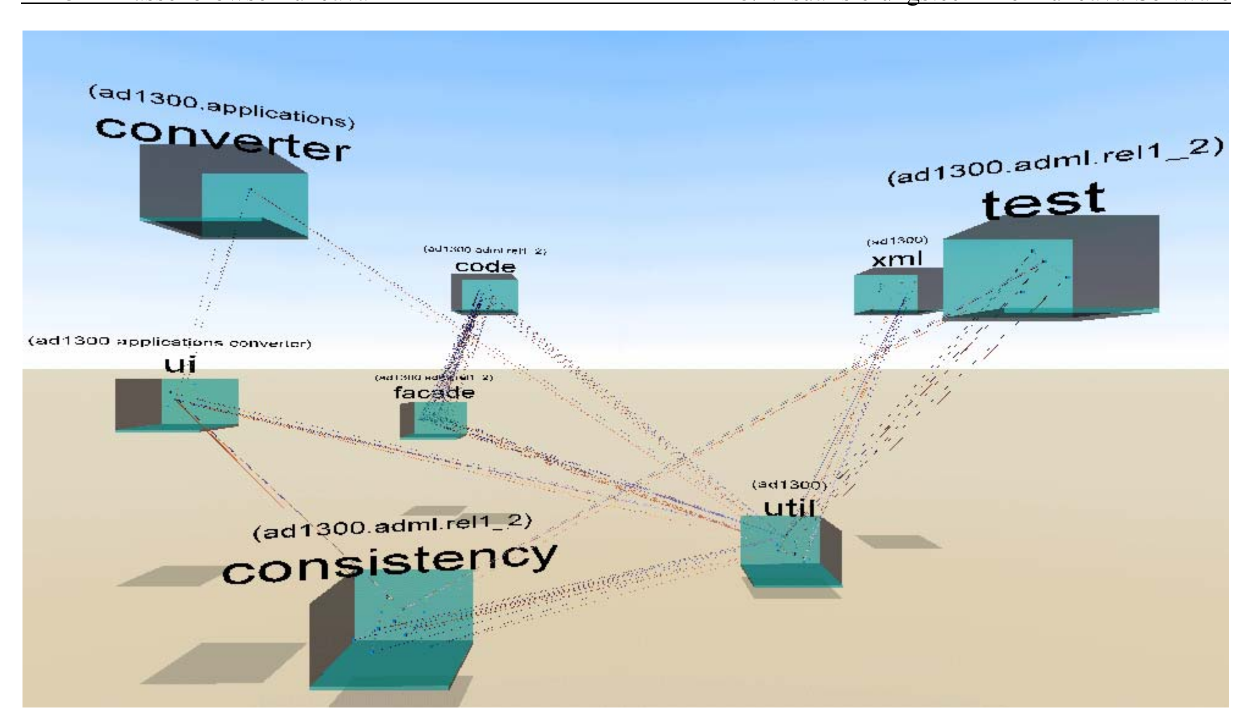


Abbildung 8.19: Beziehungen zwischen Entitäten verschiedener Pakete

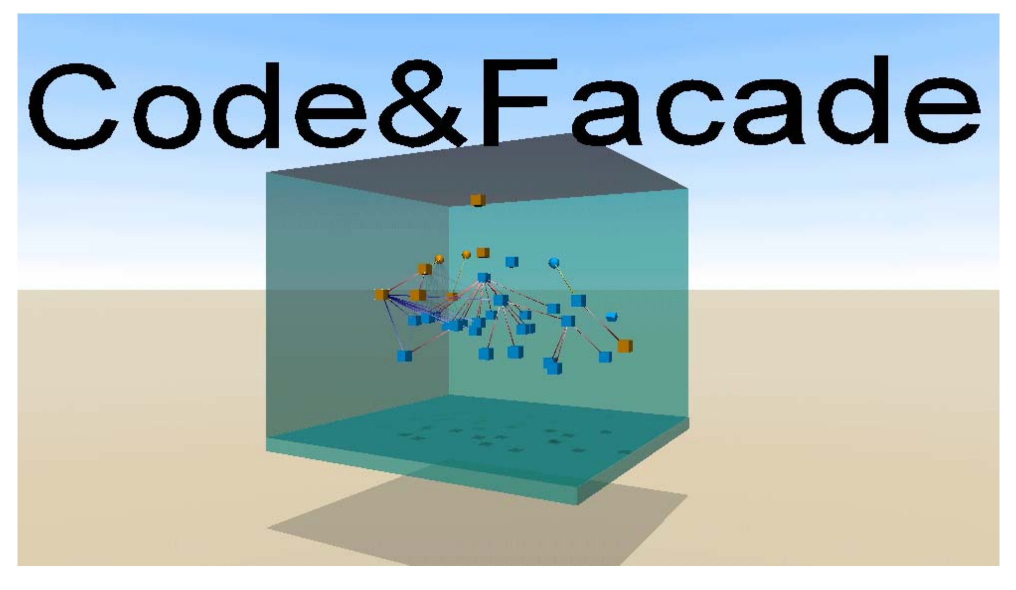


Abbildung 8.20: Zusammenfassung zweier Pakete bei Unterscheidung ihrer Entitäten durch die Symbolfarbe

8.4.1 Paketzusammenfassungen

Neben dem Problem der langen Pfeile wird durch die abgegrenzten Bereiche für verschiedene Pakete weiterhin die Verwendung z.B. von Cone Trees für Entitäten aus unterschiedlichen Paketen verhindert. Um diesen beiden Umständen zu begegnen, kann das Zusammenfassen mehrerer Pakete, deren Zusammenspiel im Detail betrachtet werden soll, ermöglicht werden. Das so entstehende Symbol wird im folgenden als Zusammenfassungssymbol bezeichnet und gleicht dem bekannten Paketsymbolen (vgl. Abbildung 8.20). Die Paketzugehörigkeit der Entitäten muß in Zusammenfassungssymbolen allerdings auf andere Art kenntlich gemacht werden, z.B. durch die Farbgebung, die Verwendung von Texturen oder eine erweiterte Beschriftung. Die Darstellung von Importen zwischen den Entitäten kann sich an dem orientieren, was im Abschnitt 8.2.4 für Benutzungen gesagt wurde.

8.4.2 Temporäre Symboleinblendung

Wegen u.U. langer Pfeile zwischen Symbolen für Entitäten verschiedener Pakete kann es schwierig sein, Beziehungen zwischen diesen zu erkennen. Als Abhilfe hierfür wurden Paketzusammenfassungen eingeführt. Beschäftigt sich der Betrachter aber gerade mit einer speziellen Entität, so mag es ihm bereits genügen, wenn er dargestellt bekommt, zu welchen Entitäten aus anderen Paketen diese in Beziehung steht. Bildet er hierzu eine Zusammenfassung, so muß er zunächst selbst ermitteln, welche Pakete überhaupt zusammenzufassen sind. Zudem muß er sich in der erstellten Zusammenfassung erst neu orientieren. Dabei wird die interessierende Entität während der Erforschung einer Software häufig wechseln, so daß ständig neue Zusammenfassungen gebildet und alte aufgelöst werden müßten.

Es ist daher nützlich, dem Betrachter über die mit einer ausgewählten Entität in Beziehung stehenden Entitäten zu informieren, ohne daß extra eine Zusammenfassung erzeugt werden muß. Dies kann dadurch geschehen, daß Symbole, die mit einem ausgewählten Entitätssymbol durch Pfeile verbunden sind, temporär in die Nähe des ausgewählten Symbols verschoben werden, so daß Beziehungen leichter ablesbar werden. Bei einem Wechsel der Auswahl wird die alte Verschiebung rückgängig gemacht und eine neue erzeugt, so daß häufige Änderungen des Interessenschwerpunkts berücksichtigt werden.

Da bei Einblendungen Entitätssymbole in den Bereich eines Paket- oder Zusammenfassungssymbols gelangen, zu dem sie eigentlich nicht gehören, müssen sie von den zugehörigen Symbolen unterschieden werden, um Fehlinterpretationen zu vermeiden. Dies kann z.B. durch leichte Transparenz oder ein Ausgrauen der eingeblendeten Symbole erfolgen.

8.4.3 Anzeige von Pfeilbeschreibungen

Eine weitere, sehr einfache Möglichkeit gegen das Problem der langen Pfeile vorzugehen ist es, dem Betrachter zu gestatten, jeweils einen Pfeil auszuwählen und zu diesem dann an exponierter Stelle eine Beschreibung anzuzeigen, aus der ersichtlich wird, was für eine Beziehung der Pfeil darstellt. Dieses Vorgehen ist allerdings nur geeignet, um eine spezielle Frage nach einem Pfeil zu klären. Es wird kein Überblick über den gesamten Sachverhalt gezeigt.

8.5 Reduktion der Darstellungskomplexität

Die gleichzeitige Anzeige alle Gegebenheiten einer Software führt aufgrund der Vielzahl von Pfeilen und Symbolen schnell zu einer Überfrachtung der Darstellung. Daher werden in den folgenden drei Abschnitten Maßnahmen erläutert, wie die Anzahl der Pfeile und Symbole reduziert und wie eine abstraktere Sicht auf die Softwarestruktur präsentiert werden kann.

8.5.1 Reduktion von Pfeilen

Für gewöhnlich existieren überaus viele Beziehungen zwischen den Elementen einer Software. Insbesondere sind hier Benutzungen zu nennen, die meist weit häufiger als Erweiterungen oder Implementierungen auftreten. Die gleichzeitige Darstellung aller Beziehungen führt durch übermäßig viele Pfeile schnell zu einer Überfrachtung der Darstellung. Die gemeinsame Nutzung von Pfeilen für symmetrische Beziehungen wirkt dem nicht genügend entgegen, so daß weitere Mechanismen notwendig sind, zumal ein manuelles Bestimmen der Sichtbarkeit jedes einzelnen Pfeils vielfach zu aufwendig wäre.

Einen möglichen Ansatzpunkt zur Reduktion der angezeigten Benutzungen bieten die im Abschnitt 5.1.4 beschriebenen Zugriffsmodi für Benutzungen und deren Rangordnung. Hierdurch wird es möglich, daß der Betrachter einen minimalen Zugriffsmodus für dargestellte Benutzungen angeben kann. Somit kann die Anzeige auf die für das Gesamtverständnis wahrscheinlich relevanteren Beziehungen begrenzt werden.

Eine weitere Option zeigt das Programm NV3D auf (vgl. Abschnitt 7.2.3). Dort ist es möglich Knoten als „tot“ zu markieren. Deren Kanten werden dann nicht mehr angezeigt und die Knoten sind isoliert. Ein solcher Mechanismus kann auch hier sinnvoll eingesetzt werden, z.B. dann, wenn eine Entität zu sehr vielen Entitäten in Beziehung steht. Ist dieser Umstand einmal vom Betrachter verstanden worden, trägt eine entsprechende Isolierung der Entität zur Übersichtlichkeit der Darstellung bei. In Anbetracht der Schachtelung von Symbolen kann man die Isolierung noch weiter verfeinern. Ein Symbol kann vollständig oder teilweise isoliert sein. Bei einer vollständigen Isolierung werden sowohl Pfeile, die mit dem Symbol verbunden sind als auch nach außen dringende Pfeile innerer Symbole unterdrückt. Bei einer Teilisolierung werden nur letztere Pfeile unsichtbar gemacht. Bei Paketsymbolen bleiben dadurch bspw. die Beziehungen des Pakets sichtbar, die Anzeige der durch ihre Vielzahl u.U. störenden Beziehungen der Entitäten des Pakets wird aber unterlassen.

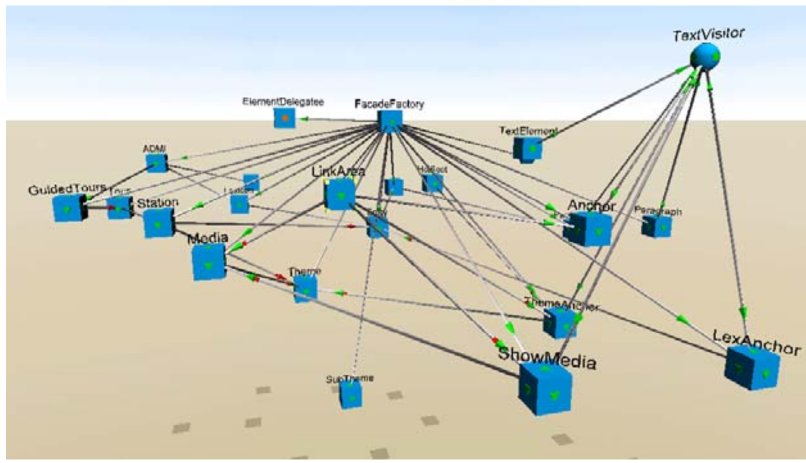
Schließlich schien es zunächst auch nützlich zu sein, eine Kombination der Filter Fish Eye View-Technik (vgl. Abschnitt 7.1.8) mit dem aus NV3D bekanntem Fading zu verwenden. Der Betrachter kann dabei durch die Auswahl einer Entität einen Brennpunkt setzen. Die globale Relevanz einer Beziehung kann sich aus deren Art und ggf. aus weiteren Eigenschaften ableiten. Bei Benutzungsbeziehung kann insbesondere auf deren Zugriffsmodus zurückgegriffen werden. Zusammen mit der Entfernung zum Brennpunkt ergibt sich so der Degree of Interest (DOI) einer Beziehung. Anstatt aber anhand des DOI binär die Sichtbarkeit des entsprechenden Pfeiles zu entscheiden, kann er durch abgestufte Helligkeits- oder Transparenzwerte unterschiedlich deutlich hervorgehoben werden.

Es zeigte sich aber, daß sich bei der Darstellung der Entitäten eines Paketes die Situation ergeben kann, daß der überwiegende Teil aller Beziehungen eine so geringe Entfernung von jedem möglichen Brennpunkt hat, daß mit der DOI-Technik kein Gewinn an Übersichtlichkeit erzielt wird, da zu viele Benutzungen einen ähnlichen DOI zugeordnet bekommen (vgl. Abbildung 8.21 auf folgender Seite).

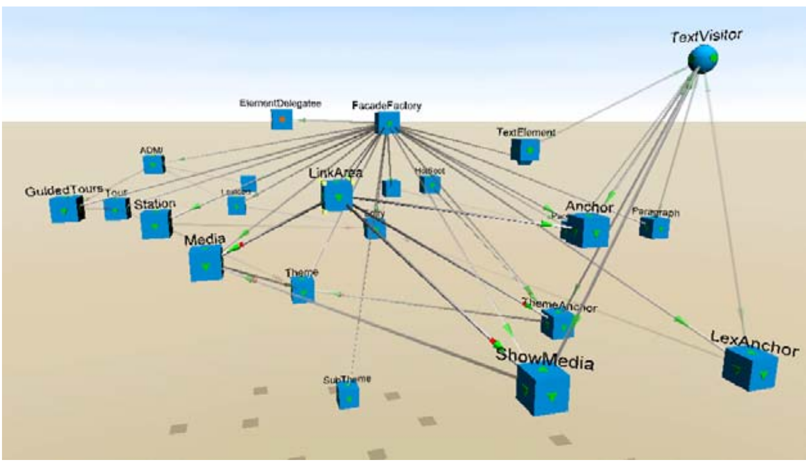
Oben beschriebene Situation ist auf die Vielzahl von Beziehungen zwischen den Entitäten eines Paketes zurückzuführen, die bei Paketen auftritt, bei denen eine starke Koheränz vorliegt. Diese Koheränz ist bei der Strukturierung eines Systems erstrebenswert, so daß die Situation recht häufig sein wird. In diesen Fällen ist es sinnvoller, lediglich die Benutzungen der im Brennpunkt liegenden Entität deutlich anzuzeigen. Alle weiter entfernten Benutzungen werden durch starke Transparenz in den Hintergrund gestellt (vgl. Abbildung 8.22 auf folgender Seite). Sie stören so nicht allzusehr, es wird aber mehr Kontext vermittelt, als wenn die Darstellung komplett unterbleibt.

Diesen Ansatz kann man dahingehend erweitern, daß man auch die Pfeilrichtung berücksichtigt. Es kann unterschieden werden, ob nur Pfeile, die auf den Brennpunkt zeigen, deutlich erkennbar sein sollen, oder nur solche, die von diesem wegweisen. Dadurch kann insbesondere die Realisierung einer symbolisierten Entität getrennt von deren Benutzung betrachtet werden. Will man die Realisierung betrachten, wird man nur die Pfeile deutlich sichtbar machen, die vom Brennpunkt abgehen, wodurch hervorgehoben wird, welche Entitäten die durch ein selektiertes Symbol gemeinte Entität benutzt, um ihre Aufgaben zu erfüllen. Umgekehrt wird sichtbar, welche Entitäten eine bestimmte Entität benutzen.

Bei der DOI-Technik ist weiterhin zu beachten, daß es bei einem gleichzeitigen Einsatz für mehrere Arten von Beziehungen dazu kommt, daß die verschiedenen Pfeile aufgrund der Transparenz schwerer zu unterscheiden sind. Die Anwendung der Technik muß daher oft auf eine Arten beschränkt bleiben. Aufgrund der besonderen Vielzahl zu erwartender Benutzungsbeziehungen bietet sich diese für die DOI-Technik an.



-a- Darstellung ohne Berücksichtigung des DOI



-b- Darstellung mit Berücksichtigung des DOI

Abbildung 8.21: Darstellung viele Benutzungen anhand der Filter-FEV-Technik mit „LinkArea“ als Brennpunkt

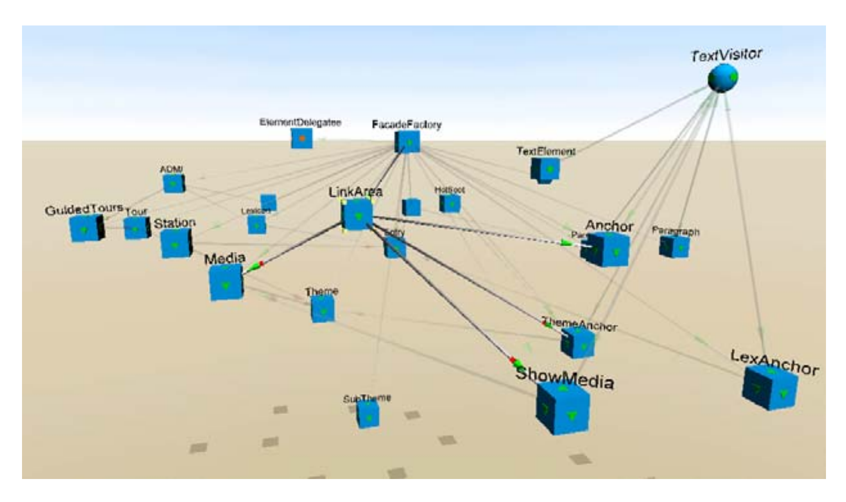


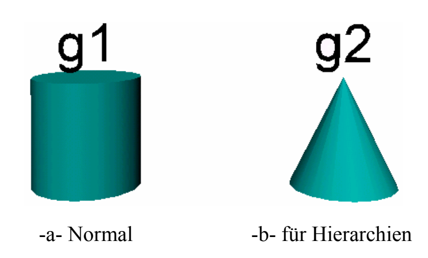
Abbildung 8.22: Darstellung viele Benutzungen durch Transparenz

8.5.2 Reduktion von Symbolen

Bereits in einem einzelnen Paket kann es zu einer Vielzahl von zugehörigen Entitäten kommen. Beispielsweise umfaßt das Paket *java.awt* der Java-API, das zur Realisierung graphischer Benutzungsoberflächen dient, 61 Entitäten. Nicht immer wird eine gleichzeitige Anzeige aller Entitäten gewünscht sein. Wie auch bei Benutzungen bietet sich zunächst eine Filtrierung der Entitätssymbole anhand ihrer Zugriffsmodi an. Die Schachtelung von inneren Entitäten in äußeren Klassen kann ebenfalls zur Reduktion genutzt werden. Aufgrund der

Tatsache, daß es sich hier um Zugehörigkeiten handelt, bietet sich eine Darstellung mittels der Information-Cubes-Technik an, die dazu führt, daß innere Entitäten optisch zurücktreten. Dabei ist es aber, wie schon bei der Paketzugehörigkeit, problematisch, daß die inneren Entitäten z.B. zu Erweiterungshierarchien gehören können, eine Verwendung von Cone Trees etc. aber durch die Schachtelung nicht mehr möglich ist. Gerade der Einsatz von Cone Trees und dergleichen bietet aber einen weiteren Ansatz zu einer Reduktion, bei der es dem Betrachter ermöglicht wird, tief in einer Hierarchie liegende und damit sehr spezielle Entitäten temporär auszublenden.

Vielfach können die Entitäten eines Pakets weiter unterteilt werden. So läßt sich beispielsweise unter den Entitäten aus *java.awt* u.a. eine Gruppe von Entitäten zur Realisierung allgemeiner Elemente einer Benutzungsschnittstelle, wie Schaltknöpfe etc., identifizieren. Eine weitere Gruppe befaßt sich mit der Anordnung einzelner Elemente von Benutzungsschnittstellen. Eine Visualisierungstechnik sollte diese Gruppenbildung unterstützen. Hierdurch könnten die Entitäten einer Gruppe, die derzeit nicht Schwerpunkt der Betrachtung sind, in einem Symbol zusammengefaßt werden, das im folgenden als Gruppensymbol bezeichnet wird. Besonders häufig können Entitäten, die eine Hierarchie, wie z.B. eine Vererbungshierarchie, bilden, als eine Gruppe angesehen werden. Zur Verdeutlichung bietet sich eine spezielle Variante des Gruppensymbols für Hierarchien an (vgl. Abbildung 8.23).



-a- Normal -b- für Hierarchien

Abbildung 8.23: Gruppensymbole

Beide Formen des Gruppensymbols verdecken die Symbole der in der jeweiligen Gruppe enthaltenen Entitäten vollständig. Hier kann von einer ikonifizierten Ansicht gesprochen werden. Sinnvollerweise können auch hier die gruppierten Symbole verkleinert in das Gruppensymbol geschachtelt werden, damit ein Betrachter sich über den Aufbau der Gruppe informieren kann, ohne die Ikonifizierung aufheben zu müssen. In der offenen Ansicht sind die Entitätssymbole deutlich sichtbar. Die Gruppenzugehörigkeit könnte wieder durch Schachtelung kenntlich gemacht werden, als flexibler hat es sich aber erwiesen, in der offenen Ansicht auf die Darstellung des Gruppensymbols zu verzichten und alle Symbole für Entitäten einer Gruppe temporär hervorzuheben, wenn eines der Symbol ausgewählt wurde. Dies kann z.B. durch die Veränderung ihrer Farbe geschehen. So werden die bei der Paketzugehörigkeit genannten Probleme der Schachtelung vermieden.

Eine Gruppierung von Paketen ist ebenfalls denkbar. Bei den betrachteten Softwarebeispielen erschien es aber nicht notwendig, Gruppen von Gruppen bilden zu können. Gegebenenfalls kann dies bei noch umfangreicherer Software aber sinnvoll sein, wobei sich das Problem der Zugehörigkeitsdarstellung dann aber verstärken würde.

8.5.3 Abhängigkeiten

Im Abschnitt 8.4 wurde eine Abbildung gezeigt, bei der anhand der Darstellung von Beziehungen zwischen den Entitäten verschiedener Pakete ein Überblick über die gesamte Software gewonnen werden konnte. Ein vergleichbarer Überblick kann dadurch entstehen, daß Abhängigkeiten zwischen den Paketen angezeigt werden, die aus den Beziehungen der Entitäten abgeleitet werden. Die Darstellung von Abhängigkeiten erlaubt die Betrachtung einer Softwarestruktur auf einem hohen Abstraktionsniveau.

Zunächst soll im folgenden geklärt werden, wann von Abhängigkeiten zu sprechen ist. Dabei braucht man sich nicht auf Abhängigkeiten zwischen Paketen beschränken; auch für Entitäten, Paketzusammenfassungen und Gruppen sind Abhängigkeiten definierbar. Obwohl

weder Paketzusammenfassungen noch Gruppen zu Strukturmodellen gehören, es sich also in diesem Sinne nicht um Elemente von Java-Software handelt, werden sie in nachfolgender Definition als solche aufgefaßt. Damit gilt für zwei Element *a* und *b*:

1) Wenn *a b* verwendet, dann ist *a* abhängig von *b*.
2) Wenn *a* abhängig von *b* ist und *a* zu einem Element *A* gehört, dann ist *A* abhängig von *b*.
3) Wenn *a* abhängig von *b* ist und *b* zu einem Element *B* gehört, dann ist *a* abhängig von *B*.

Ist ein Element *a* von einem Element *b* abhängig, so wird dies durch einen orangen Pfeil vom Symbol von *a* zum Symbol von *b* dargestellt. Es ist aber unsinnig, alle durch obige Definition entstehende Abhängigkeiten anzuzeigen. Ein Pfeil für eine aus dem ersten Satz folgende Abhängigkeit kann entfallen, da bereits der entsprechende Verwendungspfeil dargestellt werden kann. Weiterhin kann die Schachtelung von Symbolen berücksichtigt werden, so daß die Zugehörigkeit von *a* oder *b* zu Elementen *A* oder *B* im zweiten oder dritten Fall nur berücksichtigt wird, wenn diese Zugehörigkeit auch durch eine Symbolschachtelung ausgedrückt wird.

Um zu zeigen, welche Art von Beziehungen zwischen zwei in Abhängigkeit stehenden Elementen vorliegen, kann eine Erweiterung der Pfeilspitze vorgenommen werden, bei der für jede vorliegende Art der Verwendung eine Markierung am entsprechenden Abhängigkeitspfeil vorgenommen wird (vgl. Abbildung 8.24 und Abbildung 8.25). Wie bei Pfeilen wird über die Farbe einer Markierung die Art der Verwendung kodiert, wobei dieselben Entsprechungen gelten (vgl. Abbildung 8.7 auf Seite 51).

Um dabei auch einen Eindruck von der Anzahl der Beziehungen zu vermitteln, erscheint es zunächst möglich, die Dicke der Markierungen entsprechend der Beziehungsanzahl zu variieren. Dagegen spricht aber das bereits im Kapitel 8.1.1 vorgestellte Problem der schlechten Erkennbarkeit von Größenverhältnissen durch die perspektivische Verzerrung. Somit ist eine Häufigkeitsangabe über die Dicke von Markierungen nicht praktikabel und die Anzeige von Verwendungen zwischen Entitäten verschiedener Pakete durch Pfeile, wie sie im Abschnitt 8.4 gezeigt wurde, bleibt weiter nützlich.

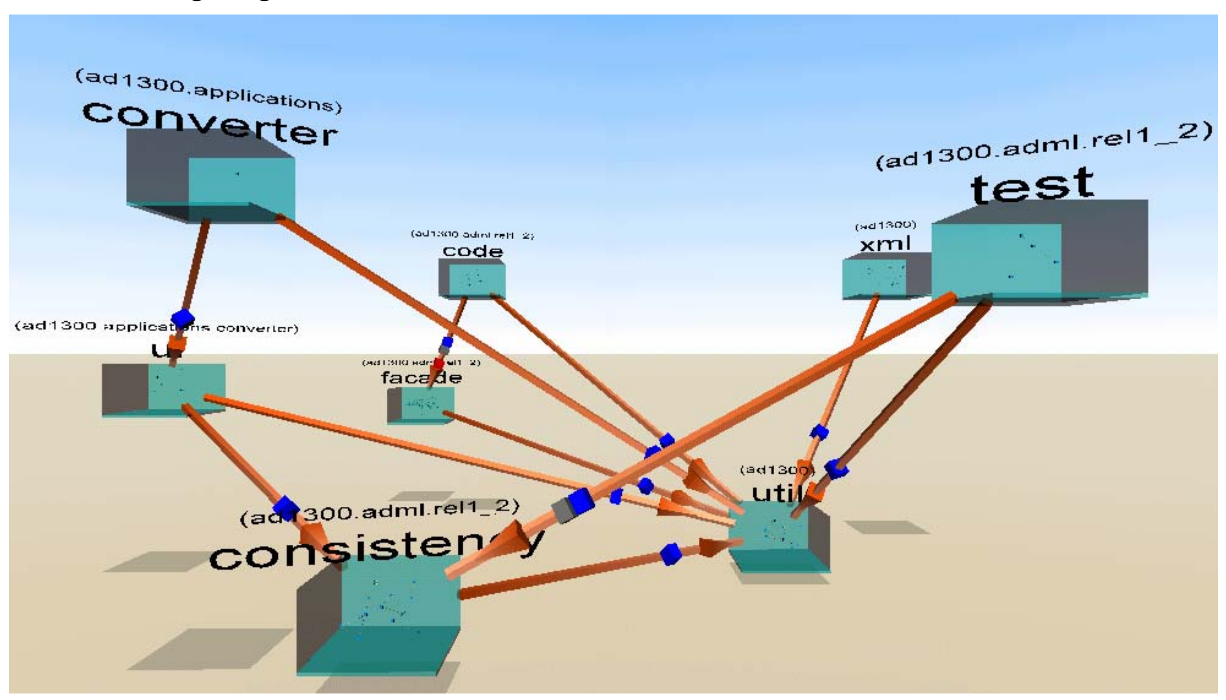


Abbildung 8.24: Darstellung der Abhängigkeiten zwischen Paketen

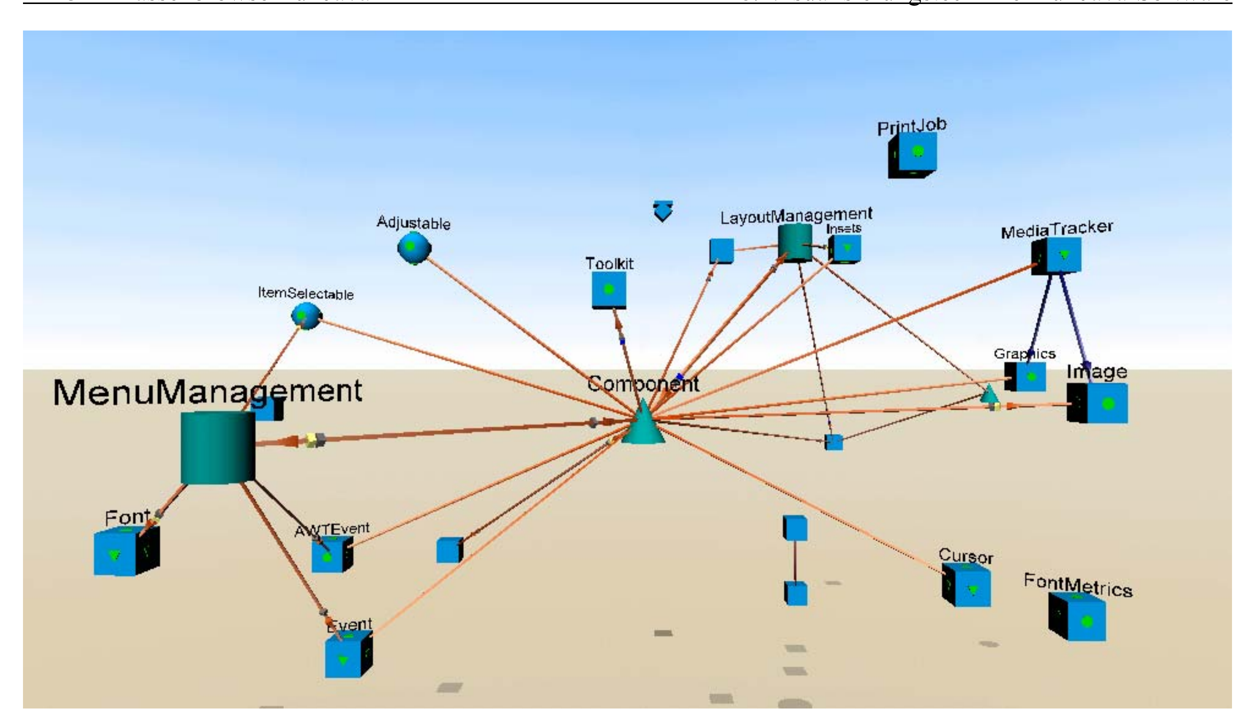


Abbildung 8.25: Beispiel für Abhängigkeiten bei Gruppen

8.6 Assoziierte Dokumente

Ohne die Verwendung zusätzlicher Dokumente und somit allein anhand einer graphischen Darstellung kann die Arbeitsweise einer Software nicht verstanden werden. Als Dokumentation bieten sich beispielsweise Entwurfsdokumente oder der Quelltext an. Bei in Java implementierter Software kann die über das zum Java Development Kit [JDK] gehörende Werkzeug Javadoc aus dem Quelltext gewonnene Dokumentation im HTML-Format hinzugezählt werden. Abbildung 8.26 zeigt ein Beispiel einer durch Javadoc generierten HTML-Seite.



Abbildung 8.26: Beispiel einer durch Javadoc generierten HTML-Seite

Zur Anzeige solcher verschiedenartigen Dokumente existieren i.d.R. bereits Werkzeuge. Hat der Betrachter in der dreidimensionalen Darstellung ein Element identifiziert über das er ein Dokument einsehen möchte, so muß er ein solches Werkzeug aktivieren. Geht man von der Verwendung einer herkömmlichen fensterorientierten Benutzungsoberfläche des Betriebssystems aus, so wird damit ein Wechsel des aktuellen Bildschirmfensters verbunden sein, der, da der auf dem Bildschirm verfügbare Platz begrenzt ist, häufig zu einer mehr oder weniger starken Überdeckung der dreidimensionalen Darstellung führen wird. Ist diese Überdeckung zu umfangreich, so wird der Betrachter beim Zurückwechseln zur Darstellung eine gewisse Zeit benötigen, sich wieder in ihr zu orientieren. Ähnlich ergeht es ihm auch beim ursprünglichen Aktivieren des Werkzeugs – auch hier ist der Wechsel mit einem Orientierungsaufwand verbunden. Bei häufigen Wechseln, die beim Erkunden eines Programmes durchaus zu erwarten sind, kann dieser Aufwand schnell als sehr störend empfunden werden. Eine geeignete Darstellung von Dokumenten muß daher als integraler Bestandteil der Visualisierung gesehen werden.

Die dreidimensionale Darstellung ermöglicht es, die Dokumente auf den Seitenwänden der Symbole darzustellen (vgl. Abbildung 8.27). Daraus ergibt sich eine natürliche Integration mit der Navigation: Um Details über ein Element zu erfahren, bewegt der Betrachter seinen Betrachtungsstandort auf das entsprechende Symbol zu, so daß der Text lesbar wird.

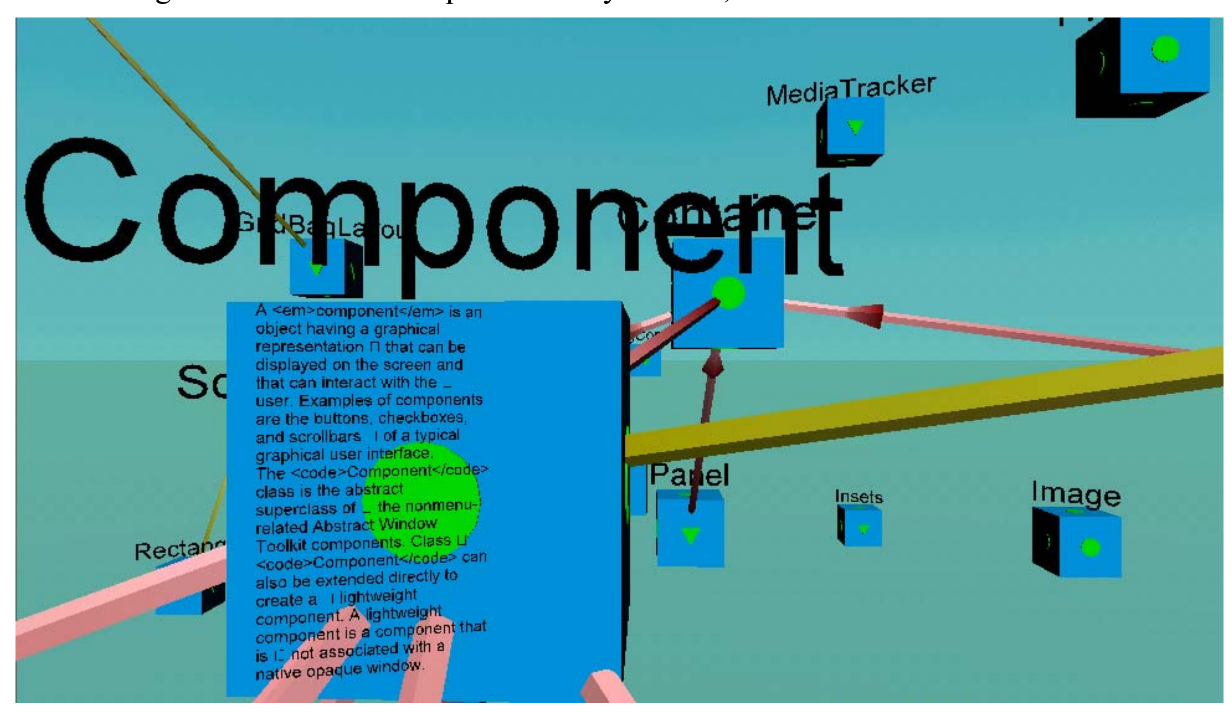


Abbildung 8.27: Darstellung von Dokumenten auf den Symbolseitenwänden

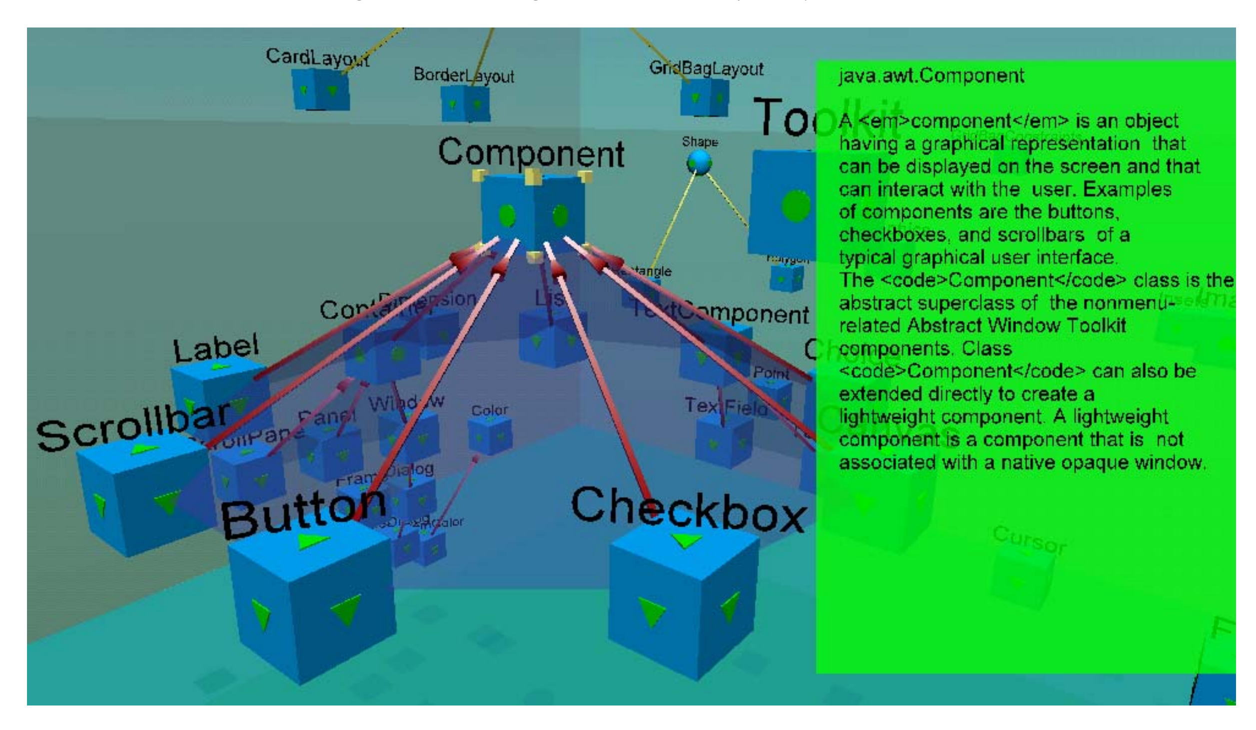


Abbildung 8.28: Darstellung von Dokumenten vor einem semitransparenten Hintergrund

Daneben hat sich die Anzeige der Dokumentation im Vordergrund der dreidimensionalen Darstellung als geeignet erwiesen. Sie erfolgt vor einem semitransparenten Hintergrund, so daß das übrige Diagramm weiterhin – wenn auch eingeschränkt – sichtbar bleibt (vgl. obige Abbildung 8.28). Gegenüber der Darstellung auf den Symbolwänden ergibt sich der Vorteil, daß der Text auch dann lesbar bleibt, wenn der Betrachtungsstandort gegenüber den Symbolen entfernt oder verdreht ist. Der Versuch, Text von den Symbolwänden abzulesen,

erfordert hingegen häufig ein mühsames Ausrichten der Betrachtungsposition und -richtung.

8.7 Mögliche Probleme bei Manipulationen

Bei der gleichzeitigen Verwendung mehrerer Visualisierungstechniken können Probleme auftreten. Eine mögliche Art von Problemen wurde bereits häufiger erwähnt: bei Schachtelungen wird durch die entstehenden abgegrenzten Bereiche die Verwendung z.B. von Cone Trees für Symbole aus unterschiedlichen Bereichen verhindert. Hier ist somit zu entscheiden, welche der beiden Techniken im konkreten Fall sinnvoller ist.

Weitere Schwierigkeiten entstehen dann, wenn dem Betrachter die Möglichkeit gegeben wird, die Darstellung zu manipulieren. Ein Beispiel für eine problematische Konstellation ist die Verwendung eines Cone Trees, der Symbole – z.B. für Entitäten – umfaßt, die alle in ein übergeordnetes Symbol – z.B. das für ein Paket – geschachtelt sind (vgl. [AF00]). Wenn der Betrachter den Cone Tree nun dreht, können Symbole des Cone Trees das übergeordnete Symbol verlassen. Wie ist dies nun zu interpretieren? Gehören die Entitäten jetzt nicht mehr zum Paket? Hat sich also die auf die dargestellte Software gestützte Semantik der Darstellung verändert? Dies wäre etwas seltsam – durch eine Manipulation, die eigentlich nur der Änderung der Ansicht dient, wäre die visualisierte Software verändert worden. Sollen solche Drehungen dann unterbunden werden? Der Erfahrung des Autors nach ist dies nicht sinnvoll: eine Drehung wird meist initiiert, um ein bestimmtes Symbol in den Vordergrund zu bringen. Würde die Drehung vorzeitig gestoppt, um zu verhindern, daß Symbole ein übergeordnetes Symbol verlassen, wäre das ausgewählte Symbol nicht vollständig in den Vordergrund gebracht und der Betrachter müßte, um Details des ausgewählten Symbols zu sehen, weitere Aktionen durchführen, z.B. den Betrachtungsstandort ändern. Dies ist umständlich und schränkt die Nützlichkeit von Cone Trees stark ein.

Manipulationen und die mit ihnen verbundene Semantik bedürfen demnach noch weiteren Untersuchungen. Ein Ansatz hierzu liefern Alfert und Fronk (vgl. [AF00]), die Manipulationen in vier Ebenen einteilen:

Ebene 0. Hier finden sich Manipulationen, die weder die Darstellung noch ihre Semantik verändern. Beispiele hierfür sind die Veränderung der Betrachterposition oder die des Betrachtungswinkels.

Ebene 1. Dies Ebene umfaßt Manipulationen, welche zwar die Darstellung verändern, aber nicht ihre Semantik. Ein Beispiel wäre eine Veränderung der Symbolfarbe, wenn diese keine spezielle Bedeutung besitzt.

Ebene 2. Manipulationen, die eine Auswirkung auf die Semantik haben bzw. haben können, wenn dies vom entsprechenden Visualisierungssystem so vorgesehen ist, werden in die Ebene 2 eingeordnet. Wenn beim Verschieben eines geschachtelten Symbols die Grenzen des übergeordneten Symbols überschritten werden, liegt z.B. eine Manipulation der Ebene 2 vor.

Ebene 3. Man kann sich auch Manipulationen vorstellen, die nicht die Darstellung betreffen, sondern den dreidimensionalen Raum, in dem sich die Darstellung befindet. Ein einfaches Beispiel wäre das Skalieren der Achsen des Raums. Derartige Manipulationen bilden die Ebene 3.

Im Rahmen dieser Arbeit konnten diesbezügliche Untersuchungen nicht mehr weiter vertieft werden. Es wurde daher vereinfachend festgelegt, daß keine Veränderung der Darstellung einen Einfluß auf die dargestellte Software hat. Aus dem Verschieben eines Entitätssymbol in ein anderes Paketsymbol folgt bspw. nicht, daß die entsprechende Entität das Paket wechselt. Damit Betrachter die Darstellung möglichst flexibel manipulieren können, werden sie zudem nicht daran gehindert, temporär auch Darstellungen zu erzeugen, die der visualisierten

Software nicht entsprechen. Diesen Umstand wieder zu beheben obliegt der Verantwortung des Betrachters. Eine Ergänzung um eine differenziertere Konzept zur Handhabung von Manipulationen, etwa im Hinblick auf ein Modellierungswerkzeug, ist aber sicher sinnvoll.

8.8 Zusammenfassung und resultierende Anforderungen

Die vorangegangenen Betrachtungen machen deutlich, daß es nicht möglich ist, eine universell einsetzbare Technik zur Visualisierung von Softwarestrukturen zu benennen. Die Eignung verschiedener Techniken ist teilweise stark von den Umständen abhängig, sowie von dem, was in der Visualisierung betont werden soll.

Für baumförmige Erweiterungshierarchien gilt, daß große Hierarchien gut durch Cone Trees visualisiert werden können, während dann, wenn man die Zusammenhänge zwischen mehreren Hierarchien zeigen will, Bäume in hintereinander liegenden Ebenen oft geeigneter sind. Der Einsatz von Kegeln ist dann nützlich, wenn zwischen den Entitäten einer Hierarchie gleichzeitig viele andersartige Beziehungen vorhanden sind oder es Entitäten außerhalb der Hierarchie gibt, die zu vielen Entitäten der Hierarchie Beziehungen haben und somit gut in den Innenraum des Kegels plaziert werden können.

Weiterhin ist die Verwendung der Information-Cubes-Technik für Pakethierarchien aufgrund der Reduktion der Darstellungskomplexität zweckmäßig. Sie bietet aber keinen sehr guten Gesamtüberblick. Bäume, Cone Trees usw. sind hier besser geeignet. Für die Paketzugehörigkeit von Entitäten ist die Schachtelung aber sinnvoll, insbesondere da sie intuitiv zu verstehen ist und einen natürlichen Übergang von einer Gesamtsicht zu Detailsichten erlaubt. Problematisch sind hier allerdings lange Pfeile zwischen den Symbolen für Entitäten verschiedener Pakete sowie das Fehlen der Möglichkeit, Cone Trees usw. paketübergreifend einzusetzen. Hier sind Paketzusammenfassungen brauchbar. Diese machen es wiederum erforderlich, die Paketzugehörigkeit auf andere – mitunter weniger intuitive – Art, wie z.B. durch Färbung, darzustellen.

Symbole für Pakete und Paketzusammenfassungen können gut in einer an die Information Landscapes erinnernden Form in der Darstellung angeordnet werden, wobei gegenüber der ursprünglichen Technik Symbole nicht auf dem Boden der Darstellung fußen müssen. Hierdurch werden Pfeilüberschneidungen vermindert. Paketzusammenfassungen sind hilfreich, um das Zusammenspiel von Entitäten verschiedener Pakete im Überblick darzustellen. Häufig genügt jedoch bereits eine Information darüber, zu welchen Entitäten aus anderen Paketen eine bestimmte Entität in Beziehung steht. Um diese Information zu vermitteln, ohne daß extra eine Paketzusammenfassung erzeugt werden muß, ist die temporäre Einblendung von Entitätssymbolen in fremde Paketsymbole und das Anzeigen von Pfeilbeschreibungen sinnvoll.

Zur Reduktion der Darstellungskomplexität sind für Symbole und Pfeile verschiedene Techniken gezeigt worden: Entitätssymbole können genau wie Benutzungspfeile nach Zugriffsmodi gefiltert werden. Die Anwendbarkeit von Degree-of-Interest-Filtern für Pfeile wird durch die häufig starke Kohärenz der Entitäten eines Pakets beeinträchtigt. Sinnvoller ist es meist, lediglich die Pfeile für Beziehungen der im Brennpunkt liegenden Entität deutlich anzuzeigen und weitere Pfeile durch starke Transparenz in den Hintergrund zu stellen. Um die Anzahl der dargestellten Symbole zu reduzieren, ist die Visualisierung von inneren Entitäten durch Schachtelung sowie die Gruppierung von Symbolen zweckdienlich. Bei der Darstellung von Hierarchien kann man es erlauben, daß Symbole für tief in der Hierarchie liegende und damit sehr spezielle Elemente temporär ausgeblendet werden.

Die Darstellung von Abhängigkeiten zwischen Elementen, Gruppen und Paketzusammenfassungen erlaubt die Betrachtung einer Softwarestruktur auf einem hohen Abstraktions-

niveau. Um allerdings die Details einer Software zu verstehen, ist die Anzeige zusätzlicher Dokumente, wie z.B. der Quelltextkommentare der Elemente, unabdingbar. Hier hat sich die Integration der Anzeige in die dreidimensionale Szene und insbesondere die Darstellung vor einem semitransparenten Hintergrund bewährt.

Die Auswahl der richtigen Visualisierungstechnik anhand der Rahmenbedingungen und der gewünschten Betonung erfordert stets auch ein gewisses Maß an Kreativität. Eine vollständige Automatisierung dieser Auswahl scheint deshalb kaum möglich. Daher ist es eine grundlegende Anforderung an ein Visualisierungssystem für Softwarestrukturen, ein großes Maß an Flexibilität zu bieten, so daß ein Benutzer mit verschiedenen Techniken experimentieren kann, bis er zu einem für ihn befriedigenden Ergebnis gelangt. Dazu muß ihm ein Satz von Techniken zur Verfügung gestellt werden, aus denen er wählen kann. Es entsteht damit eine Art „Baukasten für Visualisierungen“. Der „Baukasten“ sollte zweckmäßigerweise aus den hier vorgestellten Techniken gebildet werden, wobei Ergänzungen sicher vorstellbar sind.

Es gilt aber auch – als zweite wichtige Anforderung – den Aufwand für das Anfertigen von Visualisierungen nach Möglichkeit gering zu halten. Dies kann vor allem dadurch geschehen, daß dem Benutzer nicht zugemutet wird, Diagramme vom Grunde auf selbst anzufertigen. Vielmehr sollte ihm anfänglich eine halbwegs geeignete Ansicht präsentiert werden, die er dann nach seinen Wünschen weiter verfeinern kann – zumal auch berücksichtigt werden muß, daß der Benutzer am Anfang seiner Arbeit die zu visualisierende Software vielleicht nicht ausreichend kennt, um bereits eine genaue Vorstellung über ihrer Darstellung zu haben. Diese Vorstellung kann er sich durch das Experimentieren mit verschiedenen Techniken anhand der anfänglichen Präsentation erarbeiten. Da eine Darstellung, die zumindest annähernd top-down oder auch ebenenförmig strukturiert ist, bereits eine gute Übersicht liefern kann, und hier zudem die kombinierte Darstellung verschiedener Sprachkonzepte leicht möglich ist, bieten sich diese beiden Techniken für initiale Darstellungen an.

Im nächsten Teil dieser Arbeit wird ein Visualisierungssystem vorgestellt, in dem viele der hier gezeigten Techniken prototypisch umgesetzt wurden.

Teil III

Ein Visualisierungssystem für Java-Software

9 Überblick

Dieser Teil der Arbeit beschreibt das entwickelte Visualisierungssystem für Java-Software, dem der Name *J3Browser*, für *Java 3D Klassenbrowser*, gegeben worden ist.

Zunächst wird in diesem Kapitel das System in seinem Aufbau und seiner Arbeitsweise umrissen, wobei auch bestehende Einschränkungen genannt werden. Das Kapitel 10 beschreibt die Benutzung des System und gibt somit eine Antwort auf die dritte konzeptionelle Frage nach der Gestaltung einer Benutzungsoberfläche für ein dreidimensionales Visualisierungssystem. Die beiden letzten Kapitel dieses Teils gehen auf die prototypische Implementierung ein. Zunächst wird dabei kurz die *Virtual Reality Modeling Language* (VRML) charakterisiert, die als Basistechnologie eingesetzt wurde. Abschließend werden dann einige Entwurfsaspekte besprochen.

Die Installation des Systems wird hier nicht erläutert. Hinweise hierzu sind auf der beiliegenden CD-ROM zu finden (vgl. Anhang A). Es wird ebenfalls nicht weiter auf das Werkzeug *j3merge* eingegangen, das dazu dient, verschiedene Diagramme zu vereinen (vgl. hierzu Anhang B).

9.1 Aufbau und Arbeitsweise

In [Bro92] wird ein dreistufiges Verfahren für die Arbeit von Visualisierungssystemen beschrieben, das sich zusammenfassen läßt zu:

1) Bilden eines empirischen Modells des zu visualisierenden Sachverhaltes.
2) Aufbau einer internen Repräsentation der Visualisierung.
3) Realisierung der Visualisierung durch Ausgabe auf ein graphisches Gerät.

Auch das vorliegende Visualisierungssystem orientiert sich an diese Dreiteilung. Besonders deutlich wird dies beim Übergang von der ersten zur zweiten Stufe, da zwei verschiedene Programme eingesetzt werden. Abbildung 9.1 zeigt den groben Systemaufbau.

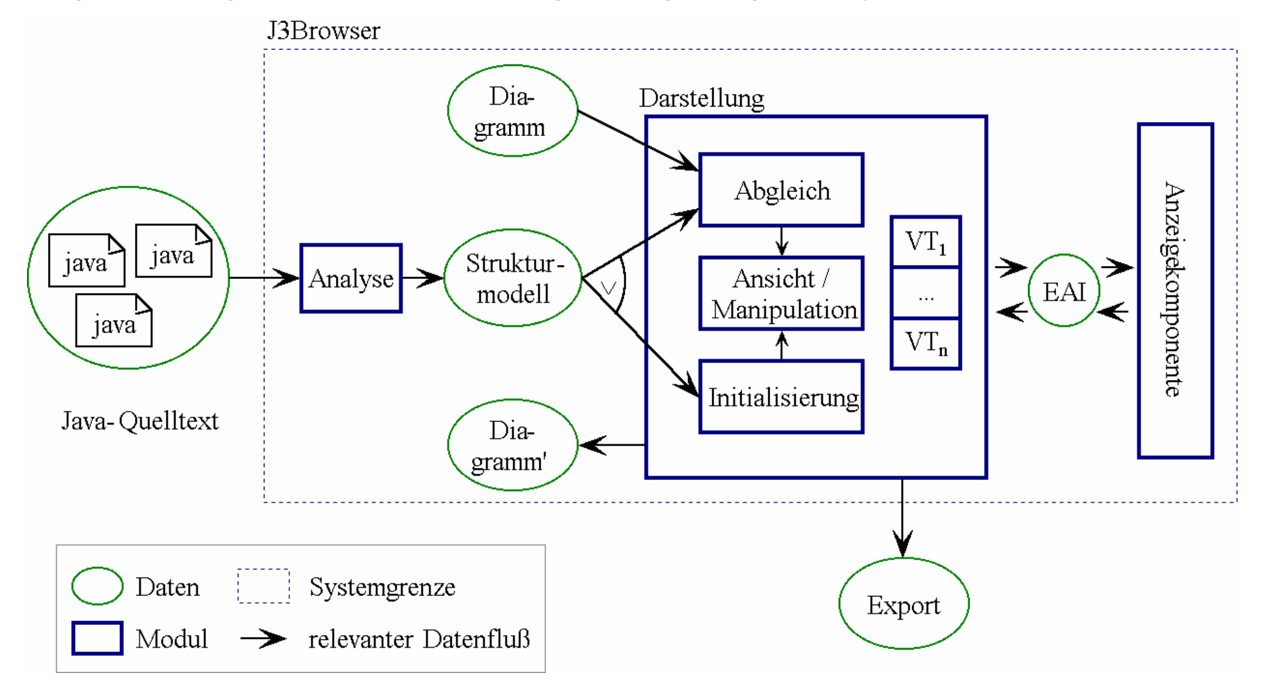


Abbildung 9.1: Grober Systemaufbau

Im ersten Schritt, der *Analyse*, wird aus einer Menge von *Java-Quelltextdateien* ein *Struktur-*

modell gewonnen, das die im Quelltext deklarierten Entitäten und Pakete sowie ihre Beziehungen beschreibt. Dieses Strukturmodell dient als Eingabe für die *Darstellung*, in der die eigentliche Visualisierung stattfindet.

Ist noch kein *Diagramm* für ein Strukturmodell vorhanden, erfolgt nun zunächst eine *Initialisierung*, bei der die interne Repräsentation einer Visualisierung aus dem Strukturmodell abgeleitet wird. Alternativ können bereits existierende Diagramme in die Darstellung eingeladen werden. Dabei erfolgt ein *Abgleich*, mit dem das eingelesene Diagramm auf eventuelle Veränderungen im Strukturmodell angepaßt wird.

Die initialisierte oder eingelesenen Visualisierung kann nun in *Ansicht* genommen werden, d.h. sie wird über eine Anzeigekomponente realisiert. Allerdings werden vor allem die gerade erst initialisierten Diagramme i.d.R. nicht hinreichend expressiv sein, so daß Benutzer des Systems aufgefordert sind, deren Ausdruckskraft durch *Manipulationen* zu verbessern. Unterstützt werden sie dabei durch eine Bibliothek von Visualisierungstechniken (in der Skizze angedeutet durch VT_1 bis VT_n). Veränderte Diagramme können gespeichert werden, was mit *Diagramm´* in der Skizze angedeutet werden soll. Zudem kann ein *Export* im VRML-Format erfolgen.

Beide verwendeten Programme, das zur Analyse und das zur Darstellung, sind in Java implementiert. Das Analyseprogramm besteht aus einer Java-Applikation, die eine graphische Benutzungsoberfläche für ein sog. *Doclet* realisiert. Doclets stehen im Zusammenhang mit dem Java-Dokumentierungswerkzeug Javadoc, welches zum Java Development Kit (JDK) [JDK] gehört. Ähnlich zu den bekannten Java-Applets, die in die Darstellung einer HTML-Seite durch einen HTML-Browser eingebunden werden, können Doclets in neueren Versionen von Javadoc (ab Version 1.2) in den Vorgang der Dokumentationserzeugung eingebunden werden. Der Vorteil dieses Ansatzes liegt darin, daß Doclets durch Javadoc bereits eine weitgehend aufbereitete Sicht auf den Quelltext haben. Javadoc stellt Methoden zur Verfügung, die es bspw. erlauben, auf einfachem Weg, die in einer Klasse deklarierten Methoden und die dazugehörigen Quelltextkommentare abzufragen. Die Entwicklung eines eigenen Parsers wäre bei weitem aufwendiger gewesen.

Die Darstellung wird in der Hauptsache implementiert durch ein Java-Applet, das in eine HTML-Seite eingebettet ist. Ebenfalls in diese Seite integriert ist eine VRML-Szene. Diese wird durch einen als Plugin für einen HTML-Browser vorliegenden und als *Anzeigekomponente* verwendeten VRML-Browser dargestellt. VRML-Plugin und Java-Applet können über eine standardisierte Schnittstelle, dem *External Authoring Interface* (EAI) [Mar97], miteinander kommunizieren und so Diagramme erzeugen und an ihnen Veränderungen vornehmen. Entsprechende Steuerdaten, die zwischen Applet und VRML-Plugin ausgetauscht werden, sind in der Abbildung 9.1 als *EAI* bezeichnet.

Für die Verwendung eines VRML-Browser spricht insbesondere, daß dieser Navigationsmöglichkeiten zum Durchwandern der dreidimensionalen Szene bereitstellt, die so nicht eigenständig entwickelt werden müssen. Weiterhin zeichnet sich insbesondere der verwendete VRML-Browser *Cosmo-Player* [COSMO] durch eine gute Darstellungsqualität aus. Zudem konnten mit VRML flexibler Ideen ausprobiert werden, als bspw. bei der Benutzung einer Klassenbibliothek, wie z.B. der Java 3D API, da ein höherer Abstraktionsgrad geboten wird. Schließlich erheben die verwendeten Komponenten den Anspruch auf Plattformunabhängigkeit, womit diese auch für das Visualisierungssystem erzielbar zu sein schien.

9.2 Einschränkungen

Während der Analyse wird ermittelt, welche Elemente der zu visualisierenden Java-Software existieren und welche Beziehungen zwischen diesen bestehen. Mittels Javadoc können fast

alle im Metamodell für Strukturmodelle berücksichtigte Arten von Elementen und Beziehungen im Quelltext gefunden werden. Es gibt jedoch eine Ausnahme. Es werden zwischen Entitäten keine Benutzungen zur Implementierung gefunden, d.h. beispielsweise solche nicht, die nur lokal innerhalb der Implementierung einer Methode erfolgen (vgl. Abschnitt 5.7). Nur Benutzungen zur Variablen-, Methoden- und Konstruktordeklaration werden gefunden.

Diese Einschränkungen scheint für die vorliegende – auch in anderer Hinsicht prototypische – Implementierung akzeptabel, hat aber die Konsequenz, das Benutzungen nur eingeschränkt visualisiert werden. Dies könnte zunächst als schwerwiegend gesehen werden, es ist aber anzunehmen, daß sich die für ein Verständnis des visualisierten Quelltext notwendigen Benutzungen vielfach in Parameterübergaben und Variablendeklarationen niederschlagen. Zudem besteht, zumindest wenn die benutzte und die benutzende Entität zu unterschiedlichen Paketen gehören, häufig zusätzlich eine Importbeziehung zwischen diesen, die dann dargestellt werden kann.

Die Darstellung der Dokumentation zu Elementen erfolgt primär im Vordergrund der dreidimensionalen Darstellung, wie sie im Kapitel 8.6 beschrieben wurde. Eine Darstellung auf den Symbolen läßt sich nur durch eine Manipulation an einer zur Implementierung gehörenden VRML-Datei erreichen, die im Anhang C beschrieben wird. Der Grund hierfür ist neben der besprochenen geringen Zweckmäßigkeit ein erhöhter Ressourcenbedarf, den ein Vorhalten dieser Option mit sich bringt. Als weitere Einschränkung werden die Symbole einer ikonifizierten Gruppe nicht in das Gruppensymbol geschachtelt. Zudem wird das Vorhandensein von Redefinitionen bei Erweiterungen während der Analyse nicht berücksichtigt und daher auch nicht dargestellt.

Javadoc kann in der vorliegenden Version Pakete nicht verarbeiten, die keine Entitäten enthalten. Dies gilt auch, wenn diese Pakete Subpakete umfassen. Dementsprechend können solche Pakete nicht mit dem J3Browser analysiert werden. Als Abhilfe bietet es sich z.B. an, derartigen Paketen Dummy-Klassen hinzuzufügen.

Die Initialisierung von Diagrammen ist in der derzeitigen Implementierung sehr einfach gehalten und genügt meist nicht dem im Kapitel 8 erhobenen Anspruch einen ersten Überblick zu liefern. Mittels einer Programmfunktion, die ein automatisches Ausrichten von Symbolen erlaubt, kann aber nach der Initialisierung unaufwendig eine starke Verbesserung erlangt werden.

Das System enthält derzeit keine Filter, die dem Rechnen mit Relationen bei ArchView vergleichbaren sind. Es werden aber verschiedene andere Mechanismen angeboten (vgl. Abschnitt 10.5.2).

10 Benutzung des Systems

10.1 Die Analyse

Das Analyseprogramm wird mittels des in Abbildung 10.1 gezeigten Bildschirmfensters gesteuert. Das Fenster besteht aus zwei Registerkarten mit den Überschriften *Aktuell* und *Pfade*. Auf der Registrierkarte *Aktuell* werden Einstellungen für den derzeit durchzuführenden Analysedurchlauf vorgenommen, wohingegen auf der zweiten Karte die Einstellung von Dateipfaden global für alle Analysen erfolgt.

Hauptbestandteil der Karte *Aktuell* ist eine Auflistung zu analysierender Pakete anhand ihrer vollständig qualifizierenden Namen. Dieses Auflistung kann mittels der Schaltflächen rechts bearbeitet werden. Dabei ist anzumerken, daß es wegen einer Einschränkung von Javadoc notwendig ist, Subpakete eines Pakets einzeln aufzuführen, wenn diese analysiert werden sollen. Die Nennung des übergeordneten Pakets ist allein nicht ausreichend.

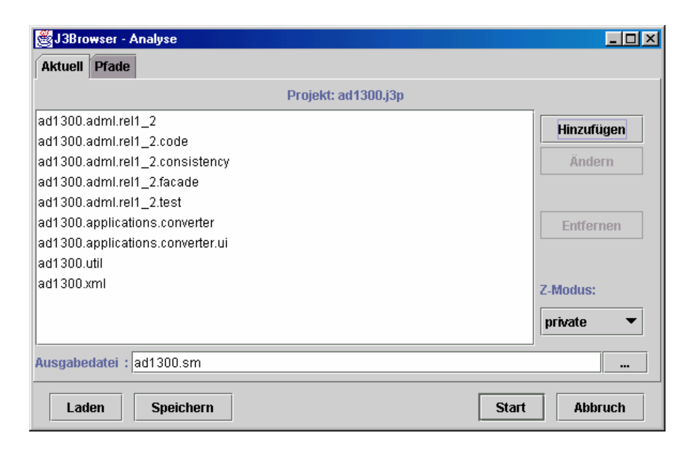


Abbildung 10.1: Das Konfigurationsfenster für die Analyse

Die aufgelisteten Pakete bilden ein sogenanntes *Projekt*. Projekte können abgespeichert und wieder eingeladen werden. Dabei führt das Laden nicht zum Löschen vorheriger Einträge, sondern neu geladene Einträge werden hinzugefügt. Dadurch können einzelne Projekte für Mengen von Paketen verwendet werden, die häufiger benötigt werden, wie z.B. für Klassenbibliotheken.

Weiterhin ist ein Kombinationsfeld vorhanden, in dem ein Zugriffsmodus gewählt werden kann. Der hier gewählte Zugriffsmodus bestimmt, wie intensiv der Quelltext durch Javadoc untersucht wird. Im Modus *public* werden z.B. nur öffentliche Entitäten und darin nur öffentliche Entitätselemente betrachtet, wohingegen *private* dazu führt, daß alle Entitäten und auch alle deren Entitätselemente untersucht werden. Dies dauert länger und das entstehende Strukturmodell ist weitaus umfangreicher. Trotzdem sollte normalerweise *private* gewählt werden und eine Filtrierung innerhalb der Darstellung erfolgen. Nur in Ausnahmefällen, z.B. bei fehlendem Speicherplatz während der Darstellungsinitialisierung, bietet sich die Verwendung anderer Modi an.

In der Zeile *Ausgabedatei* kann die Datei festgelegt werden, in der das Strukturmodell gespeichert wird. Der Knopf am rechten Rand öffnet einen Dateiauswahldialog.

In der Karte *Pfade* werden Dateipfade aufgelistet, unter denen Pakete gesucht werden. Wiederum finden sich rechts Schaltflächen über die editiert werden kann. Da die Reihenfolge der Auflistung, z.B. beim Vorliegen mehrerer Versionen des gleichen Pakets unter verschiedenen Pfaden, relevant ist, kann diese verändert werden. Die Pfade werden in einer

globalen Einstellungsdatei abgespeichert.

Mit der Anwahl der Schaltfläche *Start* beginnt der Analysevorgang. Es öffnet sich ein weiteres Fenster, in dem der Aufruf von Javadoc und die Ausgaben des Doclets angezeigt werden. Dazu ist anzumerken, daß Javadoc häufig Warnungen oder sogar „internal errors“ meldet, insbesondere dann, wenn Quelltext für Java-Versionen vor 1.2 analysiert wird. Wenn es sich dabei nicht um Fehler wegen falscher Dateipfade handelt, die an einem sofortigen Abbruch der Analyse zu erkennen sind, können diese Meldungen aller Erfahrung nach ignoriert werden.

Der Analysevorgang endet mit der Ausgabe einer Statistik über die Anzahl gefundener Elemente und Beziehungen. Erwähnt sei noch, daß die Analyse auch über die Kommandozeile gesteuert werden kann. Um Informationen darüber zu erhalten, sollte die Analyse mit der Option ‚-help‘ aufgerufen werden.

10.2 Das Hauptfenster der Darstellung

Abbildung 10.2 zeigt das Hauptfenster der Darstellung, das in ein Fenster des Netscape Communicators [NETSCAPE] eingebettet ist. Es ist unterteilt in eine Menüzeile, eine Symbolleiste, eine Statuszeile, dem Darstellungsbereich als Hauptteil und der Benutzungsoberfläche des verwendeten VRML-Browsers am unteren Rand.

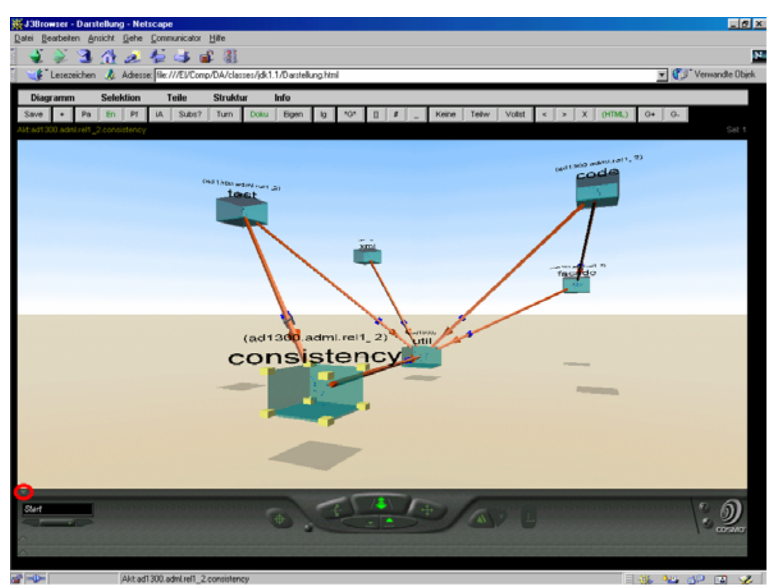


Abbildung 10.2: Das Hauptfenster der Darstellung mit roter Hervorhebung der Minimierungsschaltfläche für den VRML-Browser

Die Statuszeile wird genutzt, um den Namen von Symbolen einzublenden, die mit dem Mauszeiger berührt werden. Dies wird notwendig, da zur Steigerung der Performanz und auch um eine Überfrachtung der Darstellung zu vermeiden, die Beschriftung entfernter Symbole abgeschaltet wird. Diese wäre aufgrund der perspektivischen Verkleinerung ohnehin oft nicht lesbar. Berührt der Mauszeiger ein Symbol, tritt die Statuszeile durch eine helle Farbe deutlich hervor. Verläßt er das Symbol, wird die Farbe abgedunkelt, bis das nächste Symbol berührt wird, der Text bleibt aber unverändert. Dies führt dazu, daß der Name auch bei kleinen Symbolen, die nur schwer mit dem Mauszeiger getroffen werden, lesbar wird. Auch beim Berühren eines Pfeils wird die Statuszeile verwendet, um eine Beschreibung des Pfeil anzuzeigen.

Durch das Diagramm im Darstellungsbereich kann mit der vom VRML-Browser angebotenen Funktionalität navigiert werden. Dieser bietet dazu verschiedene Modi wie z.B. *Gehen*, *Rotieren*, *Schwenken* und so weiter. Ein Vorteil des verwendeten Cosomo-Browser ist es, daß

seine Benutzungsoberfläche minimiert werden kann, so daß mehr Platz für den Darstellungsbereich verbleibt. Da die Navigation ausschließlich über Mausbewegungen unter Zuhilfenahme der Tastatur erfolgen kann, bietet sich diese Minimierung an. Sie wird erzielt, indem das in Abbildung 10.2 rot umkreiste Dreieck angeklickt wird. Ansonsten wird im folgenden nicht weiter auf die Benutzung des VRML-Browser eingegangen, es sei auf dessen Online-Hilfe verwiesen. Erwähnt sei noch, daß sich die Aktivierung des sog. *Dauersuchmodus* bewährt hat, in dem ein Anklicken graphischer Objekte mit der rechten Maustaste dazu führt, daß sich diesem Objekt in der Art der Rapid Zooming Technik schnell genähert wird.

Nachfolgende Abschnitte beschreiben die Anwendung der Darstellung und orientieren sich dabei an den vorhandenen Menüs. Deren Aufgaben lauten im Überblick:

Diagramm: Handhabung von Diagrammen.

Selektion: Auswählen von Symbolen.

Teile: Arbeiten mit Diagrammteilen.

Struktur: Strukturieren des Diagramms.

Info: Anzeige von Informationen über den J3Browser.

Das Menü *Info* wird im folgenden nicht weiter betrachtet. Auf die Bedeutung der Symbole in der Symbolleiste wird am Ende im Abschnitt 10.7 eingegangen. Zuvor werden verschiedene Fenster vorgestellt, die dem Hauptfenster untergeordnet sind. Diese sind i.d.R. nicht-modal, so daß Benutzer des Systems sie permanent geöffnet haben können.

10.3 Handhabung von Diagrammen (Menü „Diagramm“)

Der erste Schritt bei der Arbeit mit dem Darstellungsteil des J3Browser ist es, aus einem Strukturmodell ein neues Diagramm zu erzeugen oder ein bereits vorhandenes Diagramm zu laden. Beide Funktionen werden im Menü *Diagramm* aufgerufen (vgl. Abbildung 10.3). Weiterhin finden sich hier Funktionen, um veränderte Diagramme unter gleichem oder anderem Dateiname zu speichern (*Speichern* und *Speichern unter...*). Zudem wird die Möglichkeit geboten, ein Diagramm als VRML-Datei zu exportieren, was deren Betrachtung unabhängig vom J3Browser erlaubt. Es wird aber nur ein statisches Abbild des Diagramms exportiert, d.h. Interaktion, wie z.B. Filtrierung, ist mit dem Exportergebnis nicht möglich. Schließlich kann die bereits aus der Analyse bekannte Statistik erneut abgerufen oder ein Bildschirmfenster für allgemeine Einstellungen geöffnet werden (vgl. Abschnitt 10.8).



Abbildung 10.3: Menü „Diagramm“

Strukturmodelle und Diagramme werden in verschiedenen Dateien gespeichert. Diese werden als *Strukturmodell-* bzw. *Darstellungsdateien* bezeichnet. Jede Darstellungsdatei umfaßt ein Diagramm und enthält einen Verweis auf die zugrundeliegende Strukturmodelldatei, die sich im selben Verzeichnis befinden muß. Bei einem erneuten Einladen einer Darstellungsdatei kann der Umstand eintreten, daß sich das assoziierte Strukturmodell in der Zwischenzeit durch eine erneute Analyse verändert hat, so daß ein Abgleich erforderlich wird. Veränderungen in den Beziehungen zwischen den Elementen, wie bspw. Benutzungen und Importe, sind dabei unproblematisch, da die entsprechenden Pfeile immer neu erzeugt werden, d.h. nicht Bestandteil der Darstellungsdatei sind. Zu beachten ist aber, daß Elemente hinzugekommen oder auch weggefallen sein können. Sind Elemente hinzugekommen, wird der Benutzer während des Ladevorgangs gefragt, ob entsprechende Symbole erzeugt werden sollen. Symbole für weggefallene Elemente werden automatisch gelöscht, worüber der

Benutzer informiert wird.

Umbenennungen von Elementen, zu denen auch Veränderungen wie bspw. das Wechseln der Paketzugehörigkeit von Klassen gezählt werden, da sich hierbei die vollständig qualifizierenden Namen ändern, werden als ein Wegfall des ursprünglichen Elements zusammen mit einem Hinzukommen eines neuen Elements interpretiert.

Diagramme können nicht geschlossen werden. Statt dessen ist ein Neustart des J3Browsers erforderlich. Beim Netscape Communicator erfolgt dieser sinnvollerweise durch Betätigung der Schaltfläche „Neu laden" des Communicators bei gedrückter Umstelltaste.

Vor dem Laden oder Speichern können einige Optionen festgelegt werden. So kann beim Laden bestimmt werden, ob bei Abhängigkeitspfeilen Farbmarkierungen für die vorliegenden Beziehungen erzeugt werden sollen, ob Berührungen von Pfeilen erkannt werden sollen (was zu mehr Speicherbedarf führt), ob auf den Symbolwänden Dokumente angezeigt werden sollen (nur möglich nach der im Anhang C beschrieben Änderung der Implementierung) oder ob der Ladevorgang beschleunigt werden soll (kann bei sehr großen Diagrammen zu Fehlern führen!). Beim Speichern kann eine Beschränkung auf sichtbare Symbole erfolgen.

*10.4 Auswählen von Symbolen (Menü „*Selektion*")*

Viele der Funktionen des J3Browsers beziehen sich auf zuvor vom Benutzer ausgewählte Symbole. Diese Selektion erfolgt in den meisten Fällen durch Anklicken eines Symbols in der Darstellung mit der Maus. Es werden zwei Selektionsmodi unterschieden: *Einfachselektion* und *Mehrfachselektion*. Bei aktivierter Einfachselektion ist immer genau ein Symbol selektiert, das Anklicken eines neuen Symbols führt zur Deselektion des vorher ausgewählten Symbols. Im Modus *Mehrfachselektion* wird zwischen einer Hauptselektion, die aus einem Symbol besteht, und einer Nebenselektion, die aus mehreren Symbolen bestehen kann, unterschieden. Die Anwahl eines neuen Symbols führt jetzt dazu, daß es zur Hauptselektion wird, und das zuvor hauptselektierte Symbol fortan zur Nebenselektion zählt. Ein Anklicken eines zur Nebenselektion gehörenden Symbols entfernt es aus dieser. Haupt- und Nebenselektion werden durch farbige Markierungen des Symbols angezeigt. Bei aktivierter Einfachselektion gilt das allein selektierte Symbol als Hauptselektion.

Zwischen den beiden Modi kann mittels des Selektionsmenüs (vgl. Abbildung 10.4) oder der Symbolleiste (vgl. Abschnitt 10.7) umgeschaltet werden. Ein Umschalten mittels der Tastatur, wie es in anderen Programmen häufig zu finden ist, ist nicht möglich, da das Drücken von Tasten technisch bedingt nicht abgefragt werden kann, während der Mauszeiger im Darstellungsbereich ist.

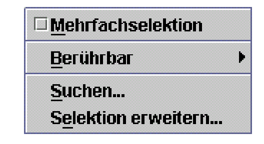


Abbildung 10.4: Selektionsmenü

Eine Vielzahl selektierbarer Symbole erschwert die Navigation, die ebenfalls u.a. mit den Maustasten gesteuert wird. Bereits die Abfrage, ob eine Berührung eines graphischen Objekts vorliegt, steht mit der Navigation im Konflikt. Deshalb kann über das Selektionsmenü und die Symbolleiste festgelegt werden, welche Arten von Symbolen selektierbar bzw. allgemeiner, welche Art von Objekten berührbar sind.

Über das Selektionsmenü können zwei untergeordnete Bildschirmfenster geöffnet werden: ein Fenster zum Suchen von Elementen und das Selektionserweiterungsfenster. Im Suchfenster kann ein vollständig qualifizierender Name eingegeben werden. Ein entsprechendes Element wird dann gesucht und dessen Symbol ggf. selektiert. Zusätzlich wird der Weg von der aktuellen Betrachterposition zum Symbol markiert.

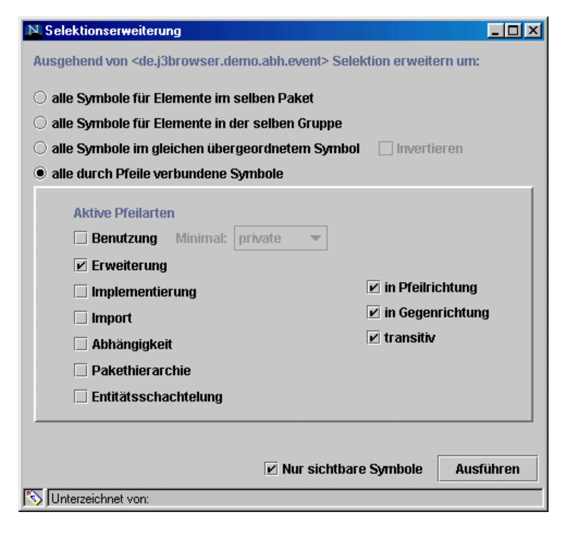


Abbildung 10.5: Fenster zur Selektionserweiterung

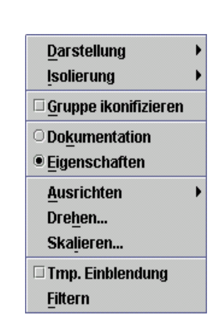


Abbildung 10.6: Teilemenü

Da das einzelne Anklicken der Symbole schnell recht mühsam werden kann, bietet die Selektionserweiterung (vgl. Abbildung 10.5) die Möglichkeit, ausgehend von der Hauptselektion die Auswahl gemäß der Struktur der visualisierten Software zu erweitern. Es können alle Symbole ausgewählt werden, die

- ein Element im selben Paket symbolisieren,
- ein Element in derselben Gruppe symbolisieren,
- oder die sich im selben übergeordnetem Symbol befinden

wie das hauptselektierte Symbol bzw. das durch dieses symbolisierte Element. Hier gilt, daß die Selektion gleichzeitig nur in einer Schachtelungstiefe erfolgt. Bei der dritten Option wird zusätzlich die Möglichkeit geboten, die vorhandene Haupt- und Nebenselektion zu deselektieren, wodurch sich eine Invertierung des Auswahlzustands der Symbole in einem übergeordneten Symbol ergibt. Weiterhin besteht noch die Möglichkeit, die Auswahl anhand verbindender Pfeile zu erweitern, wobei die Pfeilart und -richtung angegeben werden kann. Auch die transitive Fortführung dieser Erweiterung kann veranlaßt werden. Schließlich kann die Erweiterung in allen Fällen auf sichtbare Symbole begrenzt werden.

10.5 Arbeiten mit Diagrammteilen (Menü „Teile“)

10.5.1 Darstellungsformen für Elemente

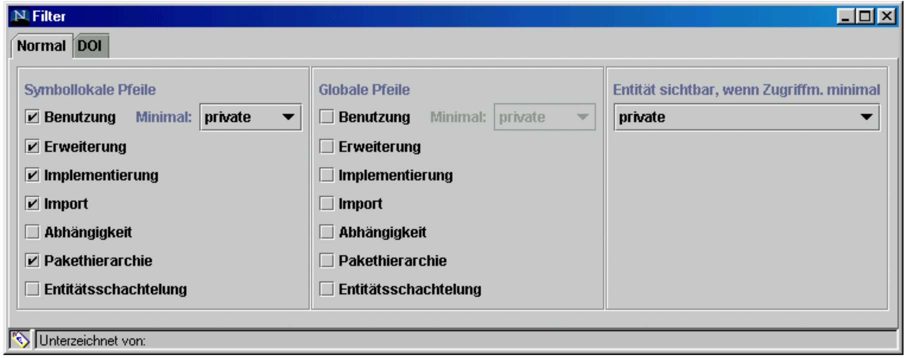
Für Symbole mit inneren Symbolen sind verschiedene Darstellungsformen vorgesehen, die über das Teilemenü ausgewählt werden können (vgl. Abbildung 10.6). Diese sind *einsehbar*, wodurch das Symbol semitransparent dargestellt wird und innere Symbole sichtbar werden, und *uneinsehbar*. Bei dieser Darstellungsform sind die Wände undurchsichtig. Bei Symbolen für Pakete und Paketzusammenfassungen besteht zusätzlich die Möglichkeit, nur die Unterseite des Symbol – genannt *Sockel* – anzuzeigen.

Unabhängig von diesen Darstellungsformen kann zur Filtrierung die Isolierung von Symbolen veranlaßt werden. Die aktuell vorliegende Isolierung eines Symbols wird auf Wunsch über dessen Beschriftung angezeigt (vgl. Anhang D).

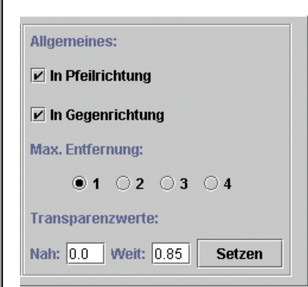
Neben der Isolation von Symbolen – und damit der Reduktion angezeigter Pfeile – kann die Komplexität der Darstellung weiter reduziert werden, indem Gruppen ikonifiziert werden, was dazu führt, daß die gruppierten Symbole in einem Symbol zusammengefaßt werden. Auch dies wird über das Teilemenü gesteuert.

10.5.2 Filter

Der Befehl *Filter* des Teilemenüs öffnet das gleichnamige Fenster. Dieses besitzt die zwei Registrierkarten *Normal* und *DOI* (vgl. Abbildung 10.7). Beide sind in drei Spalten eingeteilt, wobei die linke und mittlere Spalte für beide Karten annähernd gleich sind. Die rechte Spalte für die Karte *DOI* zeigt Abbildung 10.7b.



-a- aktivierte Karte „Normal"



-b- Veränderte rechte Spalte bei Karte „DOI"

Abbildung 10.7: Filterfenster

Es wird zwischen lokalen und globalen Pfeilen unterschieden. Ein lokaler Pfeil liegt vor, wenn beide durch den Pfeil verbundenen Symbole dasselbe übergeordnete Symbol besitzen, bzw. wenn beide kein übergeordnetes Symbol besitzen. Nicht lokale Pfeile sind global.

Auf der Karte *Normal* kann getrennt für lokale und globale Pfeile und nach Pfeilarten entschieden werden, ob entsprechende Pfeile angezeigt werden sollen. Wenn Pfeile für Benutzungen dargestellt werden sollen, kann zusätzlich ein minimaler Zugriffsmodus festgelegt werden, ab dem diese sichtbar sind. Ähnliches gilt für Entitätssymbole.

Nach Aktivierung der Karte *DOI* kann ein Degree-of-Interest-Filter aktiviert werden – wiederum getrennt für lokale und globale Pfeile und nach Pfeilarten. In der veränderten rechten Spalte können dazu Parameter spezifiziert werden. Veränderungen im Filterfenster führen i.d.R. zu einer sofortigen Aktualisierung der Darstellung, einzig die beiden veränderbaren Transparentwerte für nahe und maximal entfernte Pfeile müssen über die Schaltfläche *Setzen* bestätigt werden. Zwischenwerte werden interpoliert.

10.5.3 Temporäre Einblendung

Im Kapitel 8.4 wurde das Problem besprochen, daß für das Erkennen von Beziehungen zwischen Entitäten unterschiedlicher Pakete besteht. Als eine Möglichkeit zu dessen Linderung wurde das temporäre Heranbringen von Symbolen, die mit einem ausgewählten Symbol über Pfeile verbunden sind, an das ausgewählte Symbol vorgeschlagen. Beim J3Browser kann dieser Vorgang über den entsprechenden Befehl des Teilemenüs aktiviert werden. Bei jeder Änderung der Hauptselektion werden dann Symbole in die Nähe des hauptselektierten Symbols gebracht, die durch Pfeile von einer Art mit diesem verbunden sind, für welche die Sichtbarkeit bei Symbollokalität gefordert wurde (vgl. vorherigen Abschnitt). Die Position dieser Symbole wird durch die automatische Ausrichtung bestimmt (vgl. Abschnitt 10.5.5).

10.5.4 Einfache Editierfunktionen

Das Teilemenü ist Ausgangspunkt für einfache Manipulationen des Diagramms. Dazu dient insbesondere das von hier zu öffnende Eigenschaftenfenster (vgl. nachfolgende Abbildung 10.8), in dem graphische Eigenschaften von Symbolen bestimmt werden können, wozu die Registrierkarte *Symbol* dient. In der zweiten Karte *Pfeile* kann die Sichtbarkeit von Pfeilen

bestimmt werden, die mit den selektierten Symbolen verbunden sind. Dies ist dann nützlich, wenn andere Akzentuierungen der Beziehungen gesetzt werden sollen, als über die Filtermechanismen erreichbar sind. Allerdings ist hiermit ein gewisser Aufwand verbunden. Das Eigenschaftsfenster kann auch über einen Doppelklick auf ein Symbol geöffnet werden (vgl. aber Abschnitt 10.5.6).

Zu den hier veränderbaren Eigenschaften von Symbolen zählen deren Position, Größe und Farbe. Weiterhin kann deren Sichtbarkeit an- oder abgeschaltet werden. Auf die Möglichkeit der Verankerung wird im Zusammenhang mit der automatischen Ausrichtung im nachfolgenden Abschnitt eingegangen. Die im Eigenschaftenfenster getroffenen Einstellungen gelten für alle selektierten Symbole. Die Positionsangabe im Fenster erfolgt für das hauptselektierte Symbol. Veränderungen dieser Position werden für nebenselektierte Symbole in gleicher Richtung und mit gleichem Betrag nachvollzogen. Auch die Position innerer Symbole wird entsprechend angepaßt.

Durch die freie Wählbarkeit des Betrachtungsstandortes und -winkels ist es für den Benutzer nicht immer klar ersichtlich, in welche Richtung z.B. die X-Achse in der Darstellung verläuft. Ohne weitere Unterstützung wären Positionierungen und auch Größenangaben für ihn somit schwierig. Deshalb wird, wenn das Eigenschaftenfenster aktiv ist, innerhalb der dreidimensionalen Darstellung eine Markierung an der gewählten Position und mit der gewählten Größe angezeigt, die bei Veränderungen sofort aktualisiert wird. Für die selektierten Symbole werden Änderungen erst dann gültig, wenn die Schaltfläche *Setzen* betätigt wird. Bis dahin können sie über *Zurücksetzen* rückgängig gemacht werden.

Im Teilemenü werden weiterhin Befehle zum *Ausrichten* angeboten. Neben der komplexen automatischen Ausrichtung (s.u.) stehen Funktionen bereit, wie sie auch bei vielen herkömmlichen graphischen Editoren Verwendung finden. Symbole können in eine Ebene gebracht, auf spezifizierbaren Kreisbögen oder Strecken gleichmäßig verteilt oder um wählbare Achsen gedreht werden. Auch hier erfolgt eine Unterstützung durch Markierungen in der Szene.

Die Größenänderung im Eigenschaftsfenster bezieht sich nur auf selektierte Symbole, eventuell vorhandene innere Symbole bleiben unverändert. Demgegenüber kann mittels der *Skalierung* eine Veränderung der Symbolausdehnung erreicht werden, bei der das Verhältnis zwischen der Größe innerer und äußerer Symbole konstant bleibt.

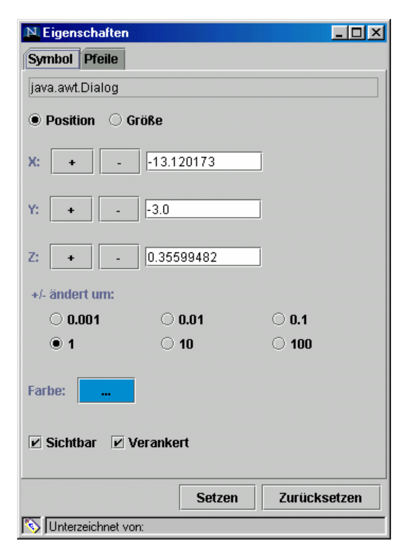


Abbildung 10.8: Eigenschaftenfenster

10.5.5 Automatische Ausrichtung

Die Gestaltung von Darstellungen mit Hilfe der im vorherigen Abschnitt beschriebenen einfachen Editierfunktionen erweist sich aufgrund der meist hohen Zahl von Symbolen schnell als mühselig. Daher verfügt der J3Browser über die Option, Darstellungen im Sinne eines Graph Drawing (vgl. Abschnitt 7.1.7) automatisch zu berechnen, bspw. um eine erste, wenig aufwendige Verbesserung der initialen Darstellung vorzunehmen. Die Editierfunktionen brauchen dann nur noch für gewünschte Verfeinerungen der Darstellung verwendet werden. Wegen der vergleichsweise häufigen Verwendung von Federmodellen kommt ein solches auch hier zum Einsatz. Es kann mittels des in Abbildung 10.9 auf nachfolgender Seite gezeigten Fensters beeinflußt werden. Die Anwendung des Federmodells ist immer begrenzt auf Symbole mit einem gemeinsamen äußeren Symbol (bzw. auf die Symbole ohne äußeres Symbol) und von gleicher Schachtelungstiefe.

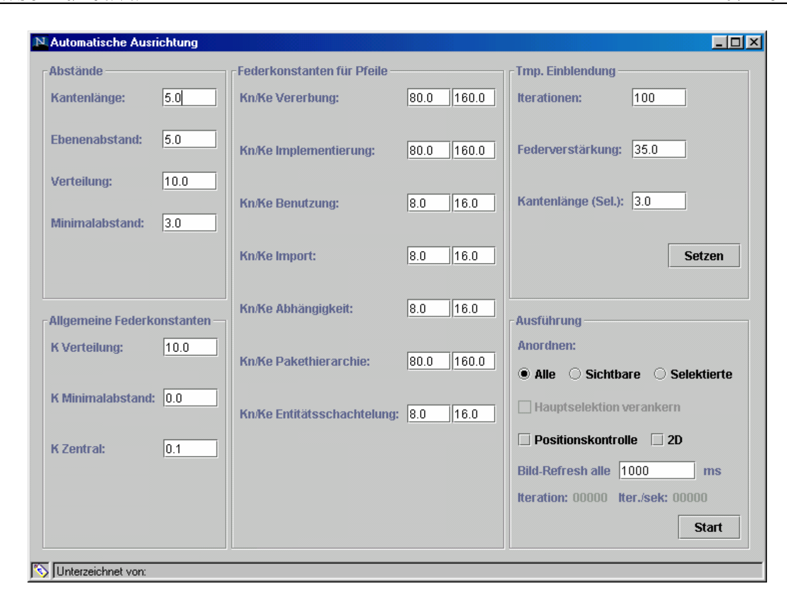


Abbildung 10.9: Fenster für die automatische Ausrichtung

Das Fenster gliedert sich in die Bereiche *Abstände*, *allgemeine Federkonstanten*, *Federkonstanten für Pfeile*, *temporäre Einblendung* und *Ausführung*.

Im ersten Bereich ***Abstände*** geht es darum, gewünschte Abstände zwischen Symbolen festzulegen. Die Kantenlänge bestimmt dabei, welchen Abstand durch Pfeile verbundene Symbole gewünschtermaßen haben sollen. Der Ebenenabstand legt den gewollten Abstand der Y-Koordinaten von Symbolen bei ebenenförmiger Darstellung bzw. bei top-down Darstellungen fest. Die Verteilung beeinflußt den Abstand zwischen unverbundenen Symbolen. Um Symbolüberschneidungen zu vermeiden, kann ein Minimalabstand zwischen Symbolen angegeben werden, bei dessen Unterschreitung eine besonders starke Feder für die Trennung der Symbole sorgt.

Unter ***Allgemeine Federkonstanten*** wird u.a. die Stärke von Federn für die Verteilung und den Minimalabstand bestimmt. Häufig kann die Berechnung von Federn für den Minimalabstand unterbleiben, da es auch ohne sie nicht zu Überschneidungen kommt. In diesem Fall sollte die entsprechende Federkonstante auf 0 gesetzt werden. Weiterhin kann eine Feder dimensioniert werden, die dazu dient, auszurichtende Symbole näher an den Mittelpunkt des übergeordneten Symbols zu bringen, wodurch kompaktere Darstellung entstehen.

Besonders wichtig für die Darstellung ist der mittlere Bereich im Ausrichtungsfenster. Durch die Variation der ***Federkonstanten für Pfeile*** kann die Deutlichkeit der Darstellung unterschiedlicher Beziehungsarten verändert werden. Beispielsweise führen hohe Werte für die Vererbungsbeziehung gegenüber geringen Werten für die Benutzung dazu, daß Vererbungshierarchien deutlicher hervortreten.

Für jede Art von Beziehung sind jeweils zwei Konstanten anzugeben. Die erste Konstante (*Kn*) beschreibt die Stärke der Federn, die zwischen zwei Symbolen für die gewünschte Kantenlänge sorgen sollen. Mit Hilfe der Konstante *Ke* wird die Feder zur Einhaltung des Ebenenabstandes dimensioniert. Hier sind auch negative Werte zulässig, die zu einer Umkehrung der Ausrichtung führen. Es werden dann bspw. benutzte Elemente oberhalb von benutzenden Elementen dargestellt. Je nach Stärke der Federn für den Ebenenabstand im Verhältnis zu anderen Feder ergibt sich eine mehr oder minder starke Ausprägung der Ebenenförmigkeit, wobei schwache Federn lediglich für eine top-down-Darstellung sorgen können.

Bei den einzelnen Werten für Abstände und Federkonstanten, die in das Fenster eingegeben

werden müssen, ist weniger der absolute Wert entscheidend als das Verhältnis der Werte zueinander. Die Felder des Fensters sind mit sinnvollen Werten zur Akzentuierung der Vererbungs- und Implementierungsbeziehung vorbelegt.

Während der temporären Einblendung wird das Federmodell benutzt, um Positionen für eingeblendete Symbole zu bestimmen. Der obere rechte Bereich ***temporäre Einblendung*** dient der Konfiguration dieser Benutzung. Zunächst kann hier die Anzahl der Iterationen bei der Simulation des Modells begrenzt werden. Bei der Benutzung des Federmodells zur Ausrichtung ist dies nicht notwendig (s.u.). Die Iterationszahl sollte nach der Leistungsfähigkeit der verwendeten Hardware dimensioniert werden, da während der Berechnung z.B. keine Veränderung der Selektion erfolgen kann.

Es hat sich als nützlich erwiesen, eingeblendete Symbole näher an das selektierte Symbol zu plazieren als zu anderen Symbolen, zu denen ebenfalls Pfeile existieren. Hierdurch treten die Beziehungen des selektierten Symbols deutlicher hervor. Um diesen Effekt zu erzielen, kann eine verkürzte Kantenlänge eingegeben werden. Damit diese auch von den entsprechenden Federn durchgesetzt werden kann, können sie gegenüber anderen Federn durch einen Multiplikationsfaktor verstärkt werden. Einstellungen für die temporäre Einblendung müssen über die Schaltfläche *Setzen* bestätigt werden.

Bevor der letzte Bereich besprochen werden kann, muß auf die Verankerung von Symbolen eingegangen werden. Verankerte Symbole werden vom Federmodell zwar in die Berechnung eingeschlossen, ihre Position wird aber nicht verändert. Dadurch können z.B. Symbole, die über die Editierfunktionen positioniert wurden, vor Veränderungen durch die automatische Ausrichtung geschützt werden. Ob ein Symbol verankert ist, wird im Eigenschaftenfenster bestimmt (s.o.) und kann über dessen Beschriftung kenntlich gemacht werden (vgl. Anhang D).

Der Bereich ***Ausführung*** dient dem Vollzug der automatischen Ausrichtung. Hier kann die Ausrichtung auch auf selektierte Symbole beschränkt werden, um unabhängig von der Verankerung Teile der Darstellung vor Veränderungen zu schützen. Geschieht dies, dann besteht die Option, das hauptselektierte Symbol als verankert zu betrachten. Dadurch gelingt es z.B., Symbole in der Nebenselektion gemäß ihrer Pfeile schnell an die Hauptselektion heranzuführen, was während der Gestaltung häufig sinnvoll einsetzbar ist. Das Federmodell kann weiterhin auf die Berücksichtigung der derzeit sichtbaren Symbole begrenzt werden.

Das Kontrollkästchen „2D“ schränkt die Berechnung auf die X- und Y-Koordinaten von Symbolen ein. Es dient der Verwendung des Federmodells in einer Ebene. Mittels aktivierter Positionskontrolle kann verhindert werden, daß Symbole während der Ausrichtung das umschließende Symbol verlassen.

Die Ausrichtung wird durch Anklicken der Schaltfläche *Start* begonnen, deren Beschriftung dann auf *Stop* wechselt. Während der Berechnung wird die Darstellung hin und wieder aktualisiert. Wie häufig dies geschieht, kann eingestellt werden, wobei wiederum die Hardwareleistungsfähigkeit zu berücksichtigen ist. Ist man mit dem Ergebnis der Ausrichtung zufrieden oder wünscht man sich andere Parameter, kann man die Berechnung über die Schaltfläche *Stop* anhalten. Während die Ausrichtung läuft, wird die Anzahl der bereits durchgeführten Iterationen angegeben. Zudem wird angezeigt, wieviel Iterationen pro Sekunden abgearbeitet werden.

10.5.6 Darstellen der HTML-Dokumentation zu einem Diagramm

Die Anzeige der HTML-Dokumentation, die aus einem Quelltext gewonnen wurde, erfolgt im Vordergrund des Darstellungsbereichs. Es wird jeweils die Dokumentation zum hauptselektierten Symbol angeboten. Mit der Anzeige wird begonnen, sobald der Befehl

Dokumentation im Teilemenü gewählt wurde. Sie wird mit den in Abbildung 10.10 gezeigten Schaltflächen der Symbolleiste gesteuert, die es erlauben (von links nach rechts in der Abbildung), in der Dokumentation vor und zurück zu blättern, die Darstellung zu schließen oder einige der eventuell in der Dokumentation vorhandenen HTML-Steuerzeichen zu unterdrücken.

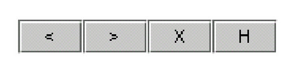


Abbildung 10.10: Schaltflächen zur Dokumentation (vgl. Text)

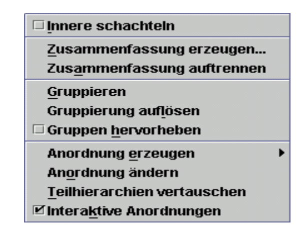


Abbildung 10.11: Strukturmenü

Wurde die Darstellung geschlossen, kann sie solange mittels Doppelklick auf ein Symbol neu geöffnet werden, bis im Teilemenü *Eigenschaften* gewählt wurde. Danach führt jeder Doppelklick wieder zum Öffnen des Eigenschaftenfensters.

10.6 Strukturieren von Diagrammen (Menü „Struktur“)

Das Strukturmenü (vgl. Abbildung 10.11) bietet dem Benutzer Möglichkeiten zur Strukturierung der Darstellung. Es dient der Benutzung der Information-Cubes-, Cone-Tree-, Baum- und Kegelvisualisierungstechnik. Weiterhin werden Funktionen für Paketzusammenfassungen und Gruppen angeboten.

10.6.1 Information Cubes

Die Information-Cubes-Technik wird über den Befehl „Innere schachteln“ gesteuert. Ist ein Symbol selektiert, das für ein Element mit inneren Elementen steht, d.h. also ein Paket mit Subpaketen oder eine Klasse mit inneren Entitäten, so können mit diesem Befehl deren Symbole in das selektierte Symbol geschachtelt werden. Wird die Information Cube Technik für ein selektiertes Symbol bereits angewendet, ist der Befehl mit einen Häkchen versehen. Seine erneute Anwahl führt dann dazu, daß die Schachtelung aufgehoben wird und – bei entsprechender Filtereinstellung – die Pakethierarchie bzw. die Entitätsschachtelung wieder über Pfeile angezeigt wird. Symbole, bei denen die Information Cube Technik angewendet wird, werden auf Wunsch durch ein der Beschriftung nachgestellten Pluszeichen kenntlich gemacht, da äußere Symbol optional auch so angezeigt werden können, daß nicht in sie hinein geblickt werden kann.

10.6.2 Paketzusammenfassungen

Paketzusammenfassungen können über den zweiten Befehl des Strukturmenüs bearbeitet werden. Besteht eine Selektion ausschließlich aus Paketen, können diese zusammengefaßt werden. Ist eine bereits existierende Zusammenfassung selektiert, kann diese wieder aufgelöst werden.

10.6.3 Gruppen

Aus selektierten Symbolen kann eine Gruppe gebildet werden. Gruppen müssen benannt werden, zudem ist das zu verwendende Symbol auszuwählen. Diese Auswahl sollte abhängig davon erfolgen, ob die selektierten Symbole eine baumförmige Hierarchie bilden - dann ist das Kegelsymbol zu verwenden – oder nicht. Im letzteren Fall wird das Zylindersymbol eingesetzt. Das Gruppensymbol ist ausschließlich für ikonifizierte Gruppen relevant. Die

Ikonifizierung wird über das Teilemenü gesteuert. Bei nicht ikonifizierten Gruppen kann die Zugehörigkeit von Symbolen über die Statuszeile ermittelt werden, da hinter dem Namen des Symbols ggf. der Name der Gruppe gezeigt wird. Weiterhin kann über *Gruppen hervorheben* veranlaßt werden, daß die Berührung eines Symbols stets dazu führt, daß alle zur selben Gruppe gehörenden Symbole hervorgehoben werden. Vorhandene Gruppen können über das Strukturmenü aufgelöst werden.

10.6.4 Anordnungen

Cone Trees, Kegel und Bäume werden unter dem Begriff der *Anordnung* zusammengefaßt. Anordnungen werden gebildet, indem das Symbol für die Wurzel der darzustellenden Hierarchie selektiert und der Befehl *Anordnung erzeugen* im Strukturmenü gewählt wird. Daraufhin erscheint ein Untermenü und der Typ der Anordnung (Cone Tree, Kegel oder Baum) kann bestimmt werden. Es erscheint ein vom Typ abhängiges Fenster, die verschiedenen Masken sind jedoch sehr ähnlich aufgebaut. Abbildung 10.12 zeigt als Beispiel die Maske für einen Cone Tree.

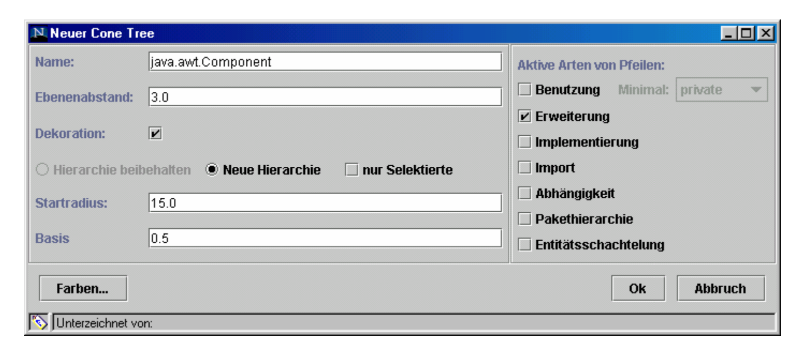


Abbildung 10.12: Erzeugen eines Cone Trees

Zuoberst kann ein Name für die Anordnung vergeben werden. Das entsprechende Feld ist mit dem Namen des Wurzelsymbols, d.h. bei neuen Anordnungen dem des hauptselektierten Symbols, vorbelegt. Weiterhin kann der Abstand zwischen den Ebenen zweier Hierarchiestufen festlegt werden. Auch besteht die Möglichkeit, zu entscheiden, ob Dekorationen erzeugt werden sollen oder nicht. Bei Dekorationen handelt es sich beispielsweise um die halbtransparenten Kegel der Cone Trees.

In der vierten Zeile ist für neue Anordnungen nur die Option *Neue Hierarchie* anwählbar. Dies bedeutet, daß der Aufbau der darzustellenden Hierarchie anhand der zwischen den Symbolen verlaufenden Pfeile ermittelt werden soll. Die Art der dabei aktiven Pfeile wird im rechten Bereich des Fensters festgelegt. Wird hier bspw. nur *Erweiterung* gewählt, besteht die erste Hierarchiestufe aus den Symbolen, von denen aus mit einem Erweiterungspfeil auf das Wurzelsymbol gezeigt wird. Es können auch Arten von Pfeilen als relevant erklärt werden, die für Beziehungen verwendet werden, die i.d.R. nicht baumförmig sind. Hier wird durch einen Breitendurchlauf ab der Wurzel künstlich eine Baumförmigkeit erzeugt. Weiterhin können auch mehrere Arten gleichzeitig aktiviert werden, um so z.B. aus Erweiterungen und Implementierungen gemischte Anordnungen zu erzeugen.

Bei Benutzungen, Importen und Abhängigkeiten wird in Pfeilrichtung durchlaufen, ansonsten in Gegenrichtung, so daß z.B. benutzte Elemente unterhalb von benutzenden Elementen dargestellt werden. Dies entspricht den im Kapitel 8 diesbezüglich gemachten Bemerkungen.

Die Hierarchiebildung ist begrenzt auf Symbole mit dem gleichen übergeordnetem Symbol (bzw. auf Symbole ohne übergeordnetem Symbol) und von einer Schachtelungstiefe. Optional kann sie weiter auf selektierte Symbole eingegrenzt werden.

Die zwei folgenden Zeilen der Maske sind für Bäume nicht vorhanden. Für Cone Trees und

Kegel kann in der oberen Zeile der Startradius bestimmt werden. Dies ist der Radius eines Kreises unterhalb des Wurzelsymbols, auf dem die Symbole der ersten Hierarchiestufe plaziert werden.

Die nächste Zeile ist für Cone Trees und Kegel verschieden. In beiden Fällen besteht ihr Zweck darin, die Radien für weitere Hierarchiestufen zu bestimmen. Bei Cone Trees hat man es mit einer Radiusabnahme und bei Kegeln mit einer Zunahme zutun. Im Falle eines Cone Trees wird die Basis einer Potenzberechnung eingegeben. Diese lautet:

$$r_n = r_1 * basis^n,$$

wobei n für die Hierarchiestufe steht und r_n für den für diese Stufe verwendeten Radius mit r_1 als Startradius. Für Kegel wird die Radiuszunahme von Stufe zu Stufe angegeben, so das sich die Formel

$$r_n = r_1 + zunahme * n$$

ergibt. Die vorgegebenen Werte haben sich für viele Hierarchien als geeignet erwiesen, müssen allerdings für besonders umfangreiche Anordnungen angepaßt werden.

Die Schaltfläche „*Farben...*“ führt zu den globalen Einstellungen. Hier können die Farben für Dekorationen global für alle vorhandenen Anordnungen getrennt nach Hierarchiestufen bestimmt werden.

Die Position der Anordnung wird durch die schon vorher gegebene Position des Wurzelsymbols bestimmt, die beibehalten wird. Wenn zwei Symbole derselben Hierarchiestufe einer Anordnung selektiert werden, können die Positionen der beiden Teilbäume durch den Befehl *Teilhierarchien vertauschen* des Strukturmenüs ausgetauscht werden.

Die beschriebenen Parameter können nachträglich verändert werden. Dazu dient der Befehl *Anordnung verändern*, der wieder zu obigen Bildschirmmasken führt. Jetzt kann auch die Option *Hierarchie beibehalten* angewählt werden, deren Vorteil es ist, daß vorgenommene Vertauschungen erhalten bleiben. Demgegenüber kann *Neue Hierarchie* genutzt werden, um Anordnungen nach einer erneuten Analyse auf geänderte Beziehungen anzupassen.

Die Position von Symbolen in Anordnungen kann weiterhin verändert werden, so daß bspw. über den Befehl *Drehen* des Teilemenüs Bäume mit anderen Ausrichtungen erzeugt werden können. Dies sollte allerdings mit Bedacht geschehen, da der optische Eindruck von Anordnungen so auch zerstört werden kann. Geschieht dies, kann über *Anordnung verändern* eine Rückführung in den Ursprungszustand erreicht werden. Symbole innerhalb von Anordnungen werden allerdings verankert, um eine Zerstörung der Anordnungen durch die automatische Ausrichtung auszuschließen.

Über *Interaktive Anordnungen* können Cone Trees und auch Kegel interaktiv gemacht werden. Nur bei interaktiven Cone Trees führt eine Selektion eines Symbols ggf. zur Drehung des Cone Trees auf den Betrachter zu. Auch Kegel werden dann bei einem Selektionswechsel um ihre Wurzel gedreht, so daß am Ende das selektierte Symbol nach Möglichkeit vor dem Betrachter steht.

Anordnungen gelten als Spezialformen von Gruppen. Daher können die Befehle *Gruppierung aufheben* und *Gruppe ikonifizieren* auch für Anordnungen angewendet werden. Für ikonifizierte Anordnungen wird immer das Kegelsymbol verwendet, da sie stets eine baumförmige Hierarchie beinhalten.

10.7 Schnellzugriff auf Funktionen über die Symbolleiste

Die Symbolleiste bietet einen gegenüber dem Menü beschleunigten Zugriff auf wichtige Funktionen des J3Browsers. Zudem sind einige Funktionen nur über die Symbolleiste

erreichbar. So kann die Geschwindigkeit, mit der sich der Betrachter durch die Szene bewegt, variiert werden. Zwar bietet der VRML-Browser dazu schon eine Möglichkeit, beim Vorliegen von tief geschachtelten Symbolen reicht diese aber nicht aus.

Abbildung 10.13 zeigt die Bedeutung der einzelnen Schaltflächen. Während der Benutzung des J3Browsers wird die Bedeutung einer Schaltfläche angezeigt, wenn sie mit dem Mauszeiger berührt wird. Einige Schaltflächen besitzen, vergleichbar mit Kontrollkästchen, zwei Zustände *an* und *aus*. Der Zustand *an* wird durch eine grüne Hervorhebung angezeigt.

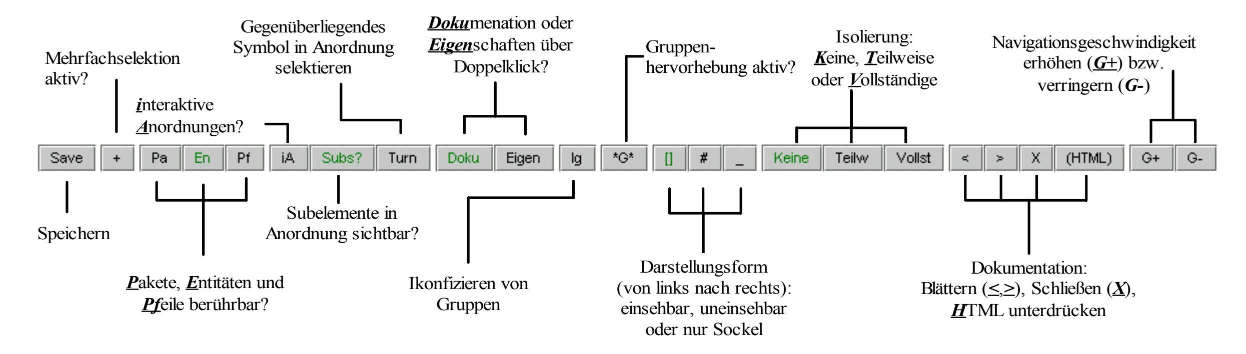


Abbildung 10.13: Symbolleiste

10.8 Globale Einstellungen (Menü „Diagramm“)

Über das Diagrammenü kann das Einstellungsfenster geöffnet werden, in dem verschiedene global gültige Optionen gesetzt werden können (vgl. Abbildung 10.15). Dazu besitzt es die Registrierkarten *Allgemein* und *Ebenenfarben*. Die Möglichkeiten der Karte *Allgemein* können der Abbildung entnommen werden. Die nicht abgebildete Registrierkarte dient dazu, Farben für Dekorationen von Anordnungen zu vergeben. Mit der Schaltfläche *Tmp. Einblendung* gelangt man zum Fenster für die automatische Ausrichtung (vgl. Abschnitt 10.5.5), in dem Einstellungen für die temporäre Einblendung vorgenommen werden können.

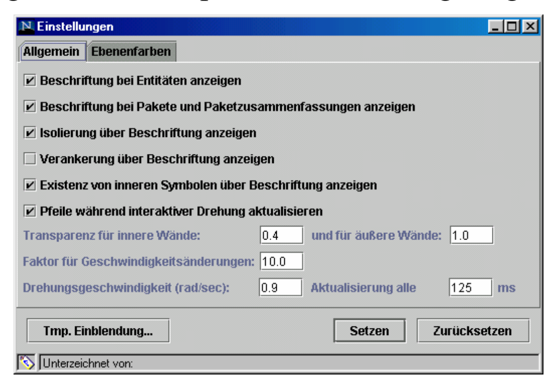


Abbildung 10.14: Fenster für globale Einstellungen

11 VRML als Basistechnologie

Nachdem im vorherigen Kapitel die Benutzung des J3Browsers beschrieben wurde, wird in diesem und im nächsten Kapitel auf dessen Implementierung eingegangen. Dieses Kapitel gibt dazu einen kurzen Überblick über die *Virtual Reality Modeling Language* (VRML), die als Basistechnologie zur Darstellung dreidimensionaler Graphiken verwendet wird. Für weitere Informationen kann beispielsweise auf [ANM97] zurückgegriffen werden.

Eine darzustellende Szene wird in VRML durch einen hierarchischen Szenengraph beschrieben, der textuell spezifiziert wird. Dazu stehen verschiedene Typen für attributierbare Knoten zur Verfügung, bspw. für Geometrie- oder Positionsangaben. Kanten werden meist implizit durch die Schachtelung von Knotendefinitionen angegeben. Folgendes Beispiel erzeugt eine Szene mit zwei roten Zylindern. Abbildung 11.1 skizziert den spezifizierten Szenengraph und Abbildung 11.2 zeigt die resultierende Darstellung.

VRML-Beispiel 11.1: [Spezifikation einer Szene]

```
#VRML V2.0 utf8
Group {
  children [
    # erster Zylinder
    Transform {
        children [
            DEF ZylinderObjekt Shape {
                geometry   Cylinder { height 0.7 radius 1 }
                appearance Appearance {
                    material Material { diffuseColor 1 0 0 }
                }
            }
        ]
        translation 0 1 0
    }

    # zweiter Zylinder
    Transform {
        children [ USE ZylinderObjekt ]
        translation 3 1 0
    }
  ]
}
```

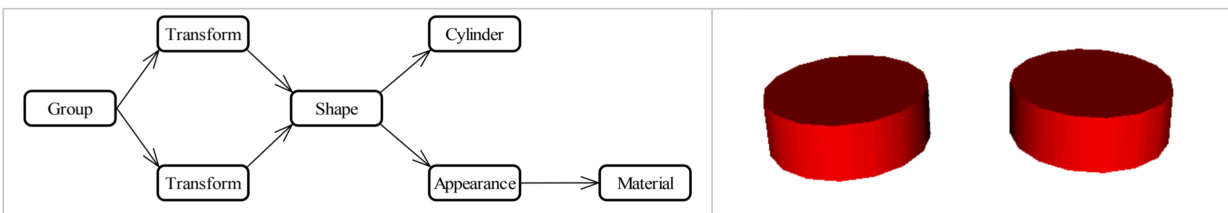


Abbildung 11.1: Skizze des Szenengraphs zum VRML-Beispiel 11.1

Abbildung 11.2: Aus VRML-Beispiel 11.1 resultierende Darstellung

Knoten vom Typ *Group* dienen der Gruppierung von untergeordneten Knoten. *Transform-*

Knoten können außer zur Positionsangabe ebenso zur Gruppierung, Skalierung und Rotation eingesetzt werden. Graphische Objekte werden durch *Shape*-Knoten beschrieben, die sich aus der Angabe einer geometrischen Form – hier *Cylinder* – und der Beschreibung des Erscheinungsbildes zusammensetzt – hier eine Rotfärbung. Das gezeigte *DEF*/*USE*-Konstrukt wird verwendet, um mehrfach auf einen Knoten zu verweisen, wodurch weitere Kanten entstehen. Ob eine Kante durch ein solches Konstrukt oder durch die Schachtelung von Knotendefinitionen spezifiziert wurde, ist für die Darstellung der Szene nicht relevant. Daher erfolgt im Szenengraph diesbezüglich keine Unterscheidung. Im Beispiel wird der Knoten für den roten Zylinder durch ein *DEF*/*USE*-Konstrukt zweifach benutzt, so daß sich bspw. eine Änderung der Farbangabe auf beide angezeigte Zylinder auswirken würde.

Neben den im Beispiel gezeigten Typen gibt es weitere. Man kann z.B. *Switch*-Knoten verwenden, um aus mehreren untergeordneten Teilgraphen einen darzustellenden auszuwählen. Während bei Switch-Knoten das Auswahlkriterium selbst bestimmt werden muß, wird bei sog. *LOD*-Knoten – für *Level of Detail* – ein anzuzeigender Teilgraph immer in Abhängigkeit zur Entfernung vom Betrachtungsstandort ausgewählt. Hierdurch können bei größeren Entfernungen geometrisch einfachere Objekte dargestellt werden, um so die Performanz der Darstellung zu erhöhen.

Um auf Aktionen des Betrachters reagieren zu können, werden *Sensor*-Knoten verwendet. Diese senden Ereignisse an wählbare Zielknoten. Als ein Zielknoten könnte z.B. ein *Switch*-Knoten dienen, der so bei Benutzeraktionen die Auswahl verändert. Unter anderem gibt es die Knotentypen *TouchSensor* und *ProximitySensor* für Reaktionen auf Mausereignisse bzw. auf Wechsel des Betrachtungsstandorts. Sensor-Knoten besitzen sog. Ereignisausgänge, die mit vorgegebenen Ereigniseingängen anderer Knoten verbunden werden können. Ereignisse, die ein Knoten an einem Ausgang erzeugt, werden an allen mit dem Ausgang verbundenen Eingänge weitergeleitet.

VRML bietet zudem die Möglichkeit, unter Verwendung bestehender Knotentypen eigene Typen abzuleiten, die dann als Prototypen bezeichnet werden. Das nachfolgende Beispiel deklariert einen Knotentyp für rote Zylinder mit wählbarem Radius und wählbarer Position, deren Anklicken zu einem Ereignis führt.

VRML-Beispiel 11.2: [Spezifikation eines Prototyps]

```
#VRML V2.0 utf8
PROTO RoterZylinder [
    field        SFFloat  r         1
    exposedField SFVec3f  position  0 0 0

    eventOut     SFBool   angeklickt
] {
    Transform {
        children [
            TouchSensor {
                isActive IS angeklickt
            }
            Shape {
                geometry   Cylinder { height 0.7 radius IS r }
                appearance Appearance {
                    material Material { diffuseColor 1 0 0 }
                }
            }
```

```
          ]
          translation IS position
      }
  }
```

Zunächst werden in einem Kopfbereich Attribute sowie Ereignisausgänge (*eventOut*) deklariert. Es wird zwischen unveränderlichen Attributen (*fields*) und Attributen, die während der Darstellung verändert werden können (*exposedFields*), unterschieden. Nach diesem Kopf folgt die Beschreibung des Aufbaus des neuen Knotentyps. Dabei wird über das *IS*-Konstrukt auf Bestandteile des Kopfes bezug genommen.

Für gewöhnlich beschreibt man den Szenengraph allein anhand einer oder mehrerer Textdateien. Um die Szene programmgesteuert zu verändern, können zwar spezielle *Script*-Knoten verwendet werden, deren Möglichkeiten sind allerdings begrenzt. Mehr Optionen bietet das *External Authoring Interface* (EAI) [Mar97]. Hier können Java-Applets zur Steuerung eingesetzt werden. Dabei ist es Voraussetzung, daß als VRML-Browser ein Plugin für einen HTML-Browser verwendet wird, und die darzustellende Szene gemeinsam mit dem steuernden Applet in einer HTML-Seite eingebettet ist. Nachdem eine Verbindung zwischen Applet und dem VRML-Browser hergestellt wurde, kann durch den Szenengraph navigiert und Attribute können verändert werden. Das Applet kann zudem als Ziel für Ereignisse fungieren. Schließlich ist es möglich, weiteren VRML-Quelltext zu erzeugen, um den die dargestellte Szene ergänzt werden kann. Gerade letzter Punkt wird für den J3Browser ausgiebig genutzt. Da bei der Verwendung des EAI zudem alle Möglichkeiten von Java zur Verfügung stehen – z.B. können komplexe arithmetische Berechnungen durchgeführt und es können graphische Benutzungsschnittstellen verwendet werden – wurde für den J3Browser das EAI anstatt Script-Knoten benutzt.

12 Realisierung des Systems

Abbildung 12.1 zeigt den Aufbau des Systems anhand der vorhandenen Paketstruktur in UML-Notation. In der Hauptsache ist das System in die Pakete *analyse*, *strukturmodell* und *darstellung* gegliedert. Das Paket *util* besitzt lediglich unterstützende Funktion, in dem es z.B. Klassen für Dateiein- und -ausgabe oder mathematische Vektoren bereitstellt. Das Paket *merge* realisiert das Werkzeug *j3merge*, mit dem Darstellungsdateien zusammengeführt werden können (vgl. Anhang B). Die Aufgaben der übrigen Pakete lauten im einzelnen:

analyse: Erzeugen von Strukturmodellen aus dem Quelltext zu visualisierender Java-Software.

strukturmodell: Speichern und Bereitstellen von Strukturmodellen.

darstellung: Graphische Darstellung von Strukturmodellen und Manipulation der Darstellung.

In den nächsten acht Abschnitten wird etwas genauer auf diese drei Pakete eingegangen. Der letzte Abschnitt dieses Kapitels bietet eine Übersicht über die Stellen der Implementierung, die der Realisierung der im Kapitel 8 vorgestellten Visualisierungstechniken dienen.

Beim Entwurf des Systems wurden einige Entwurfsmuster eingesetzt. Dabei handelt es sich um „*bewährte Lösungsideen zu immer wiederkehrenden Entwurfsproblemen*“ [Oes98, S.63]. Vielfach wurde das *Observer*-Muster eingesetzt, das es erlaubt, alle von einem Objekt abhängigen Objekte (sog. *Observer* oder auch *Listener* des Objekts) über dessen Zustandsänderungen zu informieren (vgl. [GHJ+96, S.293ff]). Für Klassen, von denen zur Laufzeit genau eine Instanz existieren soll, bietet sich das *Singleton*-Muster an (vgl. [GHJ+96, S.127ff]). Derartige Klassen werden deshalb auch als *Singleton-Klassen* bezeichnet. Eine Bezeichnung, die zur Vereinfachung in diesem Text auch für Klassen beibehalten wird, welche das Singleton-Muster nicht implementieren, aber trotzdem nur einmal im System instanziiert werden.

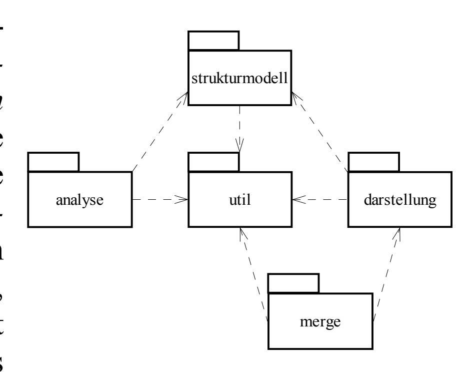


Abbildung 12.1: Paketstruktur des Systems

Das System wurde unter Verwendung der Entwicklungsumgebung *JBuilder Standard 2.0* implementiert [JBuilder]. Diese Umgebung wurde vor allem deshalb gewählt, weil sie dem Autor bereits seit längerem vertraut ist.

12.1 Paket „strukturmodell“

Das Strukturmodell einer Java-Software modelliert die im Quelltext deklarierten Elemente sowie deren Beziehungen. Das Paket *strukturmodell* enthält Entitäten, mit denen Strukturmodelle verwaltet werden. Diese sind weitgehend so aufgebaut, wie es bei der Vorstellung des Metamodells für Strukturmodell im Kapitel 5 besprochen wurde. Eine Übersicht über das Paket liefert daher die Abbildung 5.6 auf der Seite 33 dieser Arbeit.

Einige Änderungen waren für die praktische Umsetzung aber dennoch notwendig, so z.B. die Einführung eindeutig identifizierender Namen auch für Beziehungen. Hierzu wurde die entsprechende Variable *name* von *Element* nach *Gegebenheit* verschoben. Ein Beziehungsname wird durch eine Zusammensetzung der Namen der in Beziehung stehenden Elemente sowie einem Kürzel für die Art der Beziehung – z.B. „Erw“ für Erweiterungen – gebildet. Die Klasse *Strukturmodell* bietet Methoden an, mit denen Gegebenheiten anhand ihres Namens

abgefragt werden können.

Des weiteren werden die im Kapitel 5 identifizierten Eigenschaften von Gegebenheiten, die dort als Attribute oder auch Assoziationen modelliert sind, in der Implementierung durch Variablen mit dem Zugriffsmodus *private* realisiert, so daß nur durch spezielle Methoden auf diese zugegriffen werden kann. Hierdurch wird eine stärkere Kapselung erzielt. Eine Variable *kommentar* vom Typ *String* ist mit entsprechenden Zugriffsmethoden zusätzlich vorhanden, um den Quelltextkommentar von Elementen aufzunehmen.

Weiterhin existiert die Klasse *Statistik*, deren Aufgabe es ist, die nach der Analyse angezeigte und auch während der Darstellung eines Diagramms abrufbare Statistik über die in einem Strukturmodell enthaltenen Gegebenheiten zu berechnen.

Für die Klasse *Gegebenheit* und ihre Subklassen wurde ferner eine Variante des Visitor-Entwurfsmusters implementiert [GHJ+96, S331ff]. Dazu existiert die Schnittstelle *GegebenheitsVisitor*, die für jede konkrete Subklasse *X* von *Gegebenheit* eine Methode der Form *visitX*(*X g*) deklariert. In *Gegebenheit* wiederum gibt es eine abstrakte Methode *accept*(*GegebenheitsVisitor v,...*), welche in jeder konkreten Subklasse *X* so implementiert wird, daß die Methode *visitX* in *v* aufgerufen wird. So kann, ohne daß die Klasse *Gegebenheit* oder eine ihrer Subklassen verändert werden muß, neue klassenabhängige Funktionalität hinzugefügt werden, indem eine neue Implementierung von *GegebenheitsVisitor* erzeugt wird. Bei J3Browser wird dies beim Erzeugen der internen Repräsentation eines Diagramms genutzt (vgl. für Beispiele die Abschnitte 12.5.2 und 12.5.3). Die Umsetzung der Musters besitzt drei Besonderheiten:

- Die *visit*-Methoden können ein beliebiges Objekt als Ergebnis liefern. Die *accept*-Methoden reichen diese Objekte weiter.
- Die *accept*-Methoden erwarten neben dem Visitor zusätzlich einen boolschen Parameter *rekursiv*. Dieser ist nur bei Elementen von Belang. Dort gibt er an, ob nach dem Aufruf einer *visit*-Methode zusätzlich die *accept*-Methoden innerer Elemente aufgerufen werden sollen. Beispielsweise kann so ein Paket die *accept*-Methoden seiner Subpakete und seiner Entitäten aufrufen. Auf diese Weise können die Gegebenheiten eines Strukturmodells rekursiv durchwandert werden.
- Es existiert eine Klasse *GegebenheitsErwVisitor*, die *GegebenheitsVisitor* implementiert. Die Klasse besitzt auch für die abstrakten Subklassen von *Gegebenheit visit*-Methoden. Die *visit*-Methode für jede Klasse besitzt eine Vorgabeimplementierung, die darin besteht, die *visit*-Methode für die entsprechende Superklasse aufzurufen. Dadurch braucht man z.B. nur *visitElement* zu redefinieren, wenn man Funktionalität hinzufügen will, die für alle Arten von Elementen gleich ist.

Schließlich müssen Strukturmodelle gespeichert werden können. Dazu wird der in Java integrierte Serialisierungsmechanismus für Objektstrukturen genutzt. Dieser ist überaus einfach zu benutzen. Es reicht bereits aus, daß die Klassen *Strukturmodell* und *Gegebenheit* die API-Schnittstelle *Serializable* implementieren, was trivial ist, da die Schnittstelle keine Methoden deklariert.

12.2 Paket „analyse“

Dieses Paket implementiert die Analyse von Java-Software. Wie bereits erwähnt, erfolgt dies unter der Verwendung eines Doclets mittels Javadoc. Das Paket umfaßt drei wesentliche Klassen. Dazu zählt die Klasse *AnalyseFenster*, die den im Abschnitt 10.1 beschriebenen Konfigurationsdialog implementiert und als ausführbare Klasse den Startpunkt für die Analyse darstellt. Weiterhin gibt es die Klasse *AnalyseAufrufFenster*. Von hier wird Javadoc

aufgerufen. Dort wird als Doclet eine Instanz der Klasse *AnalyseDoclet* verwendet, in der über die bereitgestellte Bibliothek Informationen über den Quelltext gewonnen werden. Auf diese Weise wird ein Strukturmodell aufgebaut.

Zentraler Bestandteil der zu Javadoc gehörenden Bibliothek ist eine baumförmige Erweiterungshierarchie von Schnittstellen mit der Wurzelschnittstelle *Doc* (vgl. Abbildung 12.2). Mit Hilfe dieser Schnittstellen können Doclets Informationen über die analysierte Software abfragen. Ein Doclet bekommt bei dessen Start eine Instanz von *RootDoc*[1] übergeben. *RootDoc* kapselt die analysierte Software als Ganzes. Von hier aus kann zu weiteren *Doc*-Instanzen navigiert werden, die Eigenschaften von Paketen (*PackageDoc*), von einzelnen Klassen oder Schnittstellen (*ClassDoc*) sowie von Entitätselementen einzelner Entitäten (*MemberDoc* und Subschnittstellen) liefern.

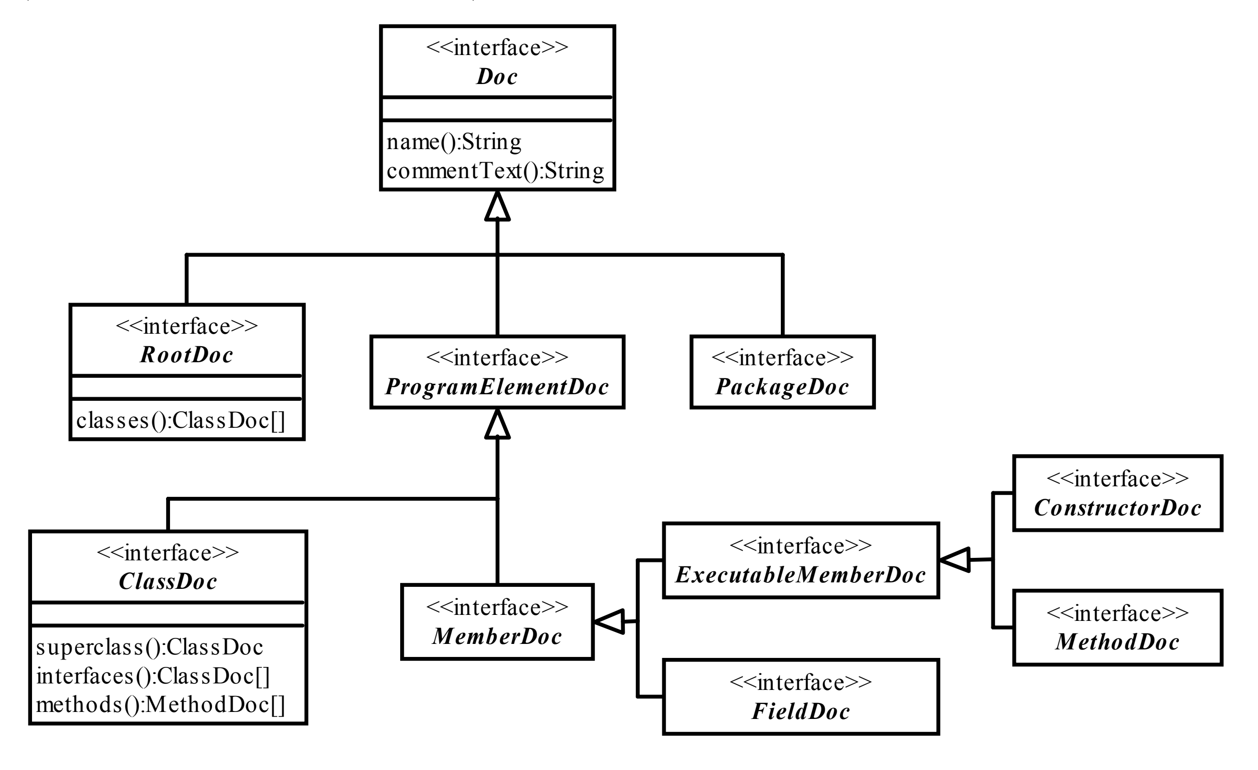


Abbildung 12.2: Doc-Schnittstellenhierarchie der Javadoc-Bibliothek

Der Aufbau eines Strukturmodells durch *AnalyseDoclet* erfolgt in zwei Schritten. Zunächst werden aus den *ClassDoc*- und *PackageDoc*-Instanzen Instanzen der entsprechenden Subklassen von *Element* gewonnen und deren Eigenschaften ermittelt, anschließend werden dann die Beziehungen zwischen den Elementen untersucht. Der erste Schritt wird durch die Methode *analysiereElemente* von AnalyseDoclet gesteuert, der zweite durch *analysiere-Beziehungen*.

Der vollständig qualifizierende Name und der Quelltextkommentar von Elementen, der Zugriffsmodus von Entitäten sowie die Instanziierbarkeit und Art von Klassen (gewöhnliche Klasse, Ausnahmeklasse oder Fehlerklasse) können direkt über entsprechende Methoden von *ClassDoc* bzw. *PackageDoc* abgefragt werden. Etwas schwieriger ist es, festzustellen, ob eine Klasse ausführbar ist. Hier muß sowohl geprüft werden, ob die Klasse von *Applet* erbt, und auch, ob eine *main*-Methode vorhanden ist. *ClassDoc* bietet die Methode *superclass* an, mit

[1] genauer: eine Instanz einer Klasse, die *RootDoc* implementiert. Die Klassen, welche die Schnittstellen der Bibliothek implementieren, sind für Doclets aber nicht sichtbar. Im Folgenden wird daher vereinfachend von Instanzen einer Schnittstelle gesprochen.

der die *ClassDoc*-Instanz für die direkte Superklasse einer Klasse abgerufen werden kann. Um festzustellen, ob eine Klasse von *Applet* erbt, werden so die Superklassen dieser Klasse rekursiv durchlaufen und dabei nach einem *ClassDoc* für *Applet* gesucht. Um zu prüfen, ob in einer Klasse eine *main*-Methode vorhanden ist, werden anhand der Methode *methods* von *ClassDoc MethodDoc*-Instanzen für alle Methoden der Klasse abgefragt. Diese werden nach einer Methode mit dem Namen *main* durchsucht, die auch den weiteren Anforderungen an eine *main*-Methode genügt: es muß sich um eine statische Methode mit dem Zugriffsmodus *public* handeln, die als Parameter ausschließlich ein Feld von Strings erwartet.

Als zweiten Schritt der Analyse werden durch die Methode *analysiereBeziehungen* die Beziehungen zwischen den gefundenen Elementen ermittelt. Auch dies geschieht meist anhand der *Doc*-Instanzen der Elemente eines Strukturmodells. So wird die bereits erwähnte Methode *superclass* der Schnittstelle *ClassDoc* genutzt, um Erweiterungsbeziehungen zwischen Klassen zu finden. Mit Hilfe der *ClassDoc*-Methode *interfaces* können sowohl die Schnittstellen gefunden werden, die eine Klasse implementiert, als auch die Schnittstellen, die eine Schnittstelle erweitert. Dementsprechend werden auf diese Art Implementierungsbeziehungen sowie Erweiterungsbeziehungen zwischen Schnittstellen ermittelt. Weitere Methoden von *ClassDoc* (*innerClasses, containingPackage* sowie *importedClasses* und *importedPackages*) werden genutzt, um Schachtelungen von Entitätsdeklarationen, Paketzugehörigkeiten von Entitäten sowie Importe von Entitäten und Paketen aufzuspüren.

Pakethierarchien allerdings werden mit Hilfe der vollständig qualifizierenden Namen von Paketen abgeleitet. Dazu wird für jedes Paket *a* eines Strukturmodells geprüft, ob im Strukturmodell ein Paket existiert, das vom Namen her das direkte Superpaket von *a* ist, also bspw. ob ein Paket *de* für ein Paket *de.darstellung* existiert. Gegebenenfalls wird dann eine entsprechende *PP*-Instanz in das Strukturmodell aufgenommen.

Die Behandlung von Benutzungen ist etwas komplexer. Für jede Entität *user* eines Strukturmodells geschieht folgendes:

- Es wird eine Liste erzeugt, die für jede Entität, die durch *user* benutzt wird, eine *Use*-Instanz umfaßt. Um die Performanz zu steigern, ist die Liste als Hash-Tabelle realisiert, bei der der Name der benutzten Entität als Schlüssel für die entsprechende *Use*-Instanz dient. Die Liste ist initial leer.
- Es werden zunächst die Variablen, dann die Methoden und schließlich ggf. die Konstruktoren von *user* durchlaufen. Anhand des Typs einer Variable, des Rückgabetyps einer Methode sowie der Parametertypen und als Ausnahmen aufgeworfenen Typen einer Methode oder eines Konstruktors werden die durch *user* benutzen Entitäten ermittelt.
- Wenn so eine durch *user* benutzte Entität *used* gefunden wurde, wird geprüft, ob in der Liste eine *Use*-Instanz für *used* existiert. Wenn nicht, wird eine solche erzeugt und sowohl in die Liste als auch in das Strukturmodell aufgenommen. Alle in einer neuen Instanz gespeicherten Zugriffsmodi sind zunächst mit *private* initialisiert, die entsprechenden Markierungen für Benutzungen als Feld oder als Variable stehen auf *false*.
- Anhand des aktuell durchlaufenen Entitätselements wird die zuvor gefundene *Use*-Instanz verändert. Wenn eine Benutzung als Feld oder als Variable vorliegt, was relativ leicht anhand der jeweiligen *MemberDoc*-Instanz festgestellt werden kann, wird die entsprechende Markierungen in der *Use*-Instanz gesetzt. Zudem wird der Zugriffsmodus des aktuellen Entitätselements mit den in der *Use*-Instanz gespeicherten Modi verglichen. Wenn für die Benutzung insgesamt oder für eine vorliegende Art der Benutzung ein im Sinne der Rangordnung aus Kapitel 5 kleiner Zugriffsmodus gespeichert ist, wird dieser durch den Zugriffsmodus des aktuellen Entitätselements ersetzt.

Nachdem beide Analyseschritte durchgeführt wurden und auf diese Art das Strukturmodell zu einer Software aufgebaut wurde, wird das Modell gespeichert, eine Statistik erzeugt und schließlich das Doclet beendet.

12.3 Paket „darstellung"

Die Darstellungskomponente des J3Browsers wird durch das Paket *darstellung* und dessen Subpakete implementiert, die in Abbildung 12.3 gezeigt werden.

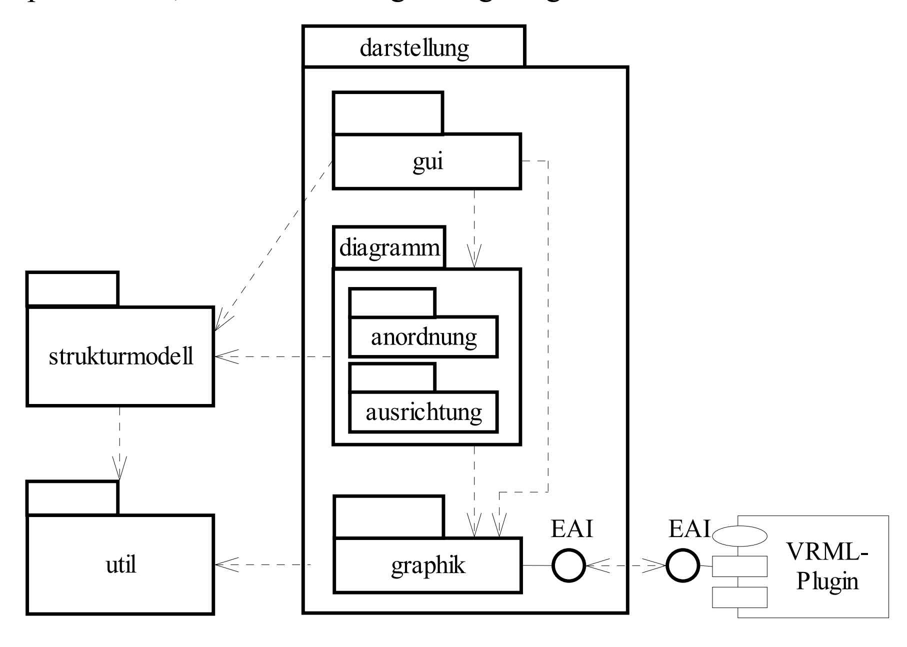


Abbildung 12.3: Aufbau des Paketes „darstellung" und dessen Einbettung ins System

Auf die verschiedenen Subpakete wird in den nachfolgenden Abschnitten näher eingegangen, zunächst sei ein Überblick über ihre Aufgaben gegeben:

gui: Realisierung der graphischen Benutzungsoberfläche.

diagramm: Interne Repräsentation von Visualisierungen sowie deren Manipulation.

diagramm.anordnung: Implementierung der Visualisierungstechniken Baum, Kegel und Cone Tree.

diagramm.ausrichtung: Enthält Ausrichtungsfunktionen, vor allem die automatischen Ausrichtung.

graphik: Realisiert die graphischen Ausgabe.

Das Paket *darstellung* selbst enthält lediglich die Singleton-Klassen *DarstellungsApplet* und *HTMLBrowser*. Erste ist die vom HTML-Browser instanziierte Startklasse des Applets. Zweite kapselt Eigenheiten des verwendeten HTML-Browsers. Dies sind insbesondere ein ggf. notwendiger spezifischer Verbindungsaufbau zum EAI und die Unterstützung des Lesens und Schreibens von Dateien durch die Anforderung von Berechtigungen beim HTML-Browser.

12.4 Paket „darstellung.gui"

Dieses Paket implementiert die im Kapitel 10 beschriebene Benutzungsoberfläche der Darstellungskomponente des J3Browsers. Dazu wird die Klassenbibliothek *Swing* verwendet (vgl. bspw. [WC98]). Das Paket umfaßt eine Singleton-Klasse für jedes komplexere

Bildschirmfenster des Darstellungsteils des J3Browsers. Diese Klassen erben alle von der Klasse *J3BFenster*, die selbst von der Swing-Klasse *JFrame* erbt. Für sehr einfache Fenster, die bspw. nur aus einer Textmeldung mit einer Ja/Nein-Frage bestehen, bietet Swing die Klasse *JOptionPane*. Diese wird nach Möglichkeit auch für den J3Browser genutzt, wobei die Verwendung durch die Klasse *MessageFenster* gekapselt wird, die für verschiedene Fragen, Meldungen usw. eigene Methoden bereitstellt.

Neben den Klassen für Bildschirmfenster umfaßt das Paket noch die Singleton-Klasse *Menu*, welche die Menüzeile zusammen mit der Symbolleiste implementiert, sowie die Singleton-Klasse *Statuszeile* und eine Reihe von Hilfeklassen, die entstanden sind, weil einige Klassen so komplex wurden, daß ihre Funktionalität weiter unterteilt werden mußte.

12.5 Paket „darstellung.diagramm“

Dieses Paket bildet den Kern der Realisierung. Es implementiert die zweite Stufe des im Kapitel 9 beschriebenen Dreistufenmodells zur Visualisierung – die interne Repräsentation von Diagrammen. Neben diesem statischen Aspekt umfaßt es auch die Veränderung der Repräsentation z.B. aufgrund vorhandener Visualisierungstechniken wie der temporären Einblendung.

Nachfolgend wird auf einige Aspekte dieses Paketes genauer eingegangen: auf die interne Repräsentation (Abschnitt 12.5.1) und deren Initialisierung (12.5.2), die Persistenz von Diagrammen und dem Abgleich mit veränderten Strukturmodellen (12.5.3), die Berechnung der Sichtbarkeit einzelner Diagrammteile (12.5.4), die optische Zusammenfassung und Trennung von Pfeilen (12.5.5), die Handhabung der Selektion und der Berührung von Diagrammteilen (12.5.6) sowie der Umsetzung der Degree-of-Interest-Darstellung (12.5.7), der Gruppierung (12.5.8) und der Manipulation von Diagrammen (12.5.9).

12.5.1 Interne Repräsentation

Die interne Repräsentation erfolgt im wesentlichen durch die Klasse *Diagramm*, die das angezeigte Diagramm kapselt, sowie durch eine Klassenhierarchie mit der Wurzel *Teil*, mittels der verschiedene Bestandteile des Diagramms – hier Symbole und Pfeile – modelliert werden (vgl. Abbildung 12.4 auf folgender Seite). Für die verschiedenartig verwendeten Symbole und Pfeile werden weitere Spezialisierungen vorgenommen (vgl. Abbildungen 12.5 und 12.6 ebenda).

Um die Information-Cubes-Technik zu realisieren, werden die Diagrammteile gemäß der Schachtelung hierarchisch gespeichert. Jedes Symbol ist über die Assoziation *innere* mit dessen inneren Teilen verknüpft und jedes Teil verfügt mit *parent* über einen Verweis auf das als *Parent* bezeichnete umschließende Symbol. Die Wurzel dieser Hierarchie bildet eine Instanz der Klasse *RootSymbol*, die ein Symbol modelliert, welches das gesamte Diagramm umfaßt (das *Rootsymbol*). Alternativ könnte man auch die Klasse *Diagramm* von *Symbol* erben lassen und das Diagramm so selbst als äußerstes Symbol auffassen, wodurch sich eine Ähnlichkeit zum *Composite*-Entwurfsmuster ergeben würde [GHJ+,S.163ff]. Da aber die Interpretation als Symbol der Klasse *Diagramm* einiges an Komplexität hinzufügen würde, wurde *RootSymbol* zusätzlich eingeführt. Aus diesem Grund wurde auch die Verwaltung einer Reihe von Einstellungen, die Diagramme betreffen, wie z.B. der Ausführlichkeit der Beschriftung, in eine eigene Klasse *DiagrammEinstellungen* ausgelagert.

Bis auf Gruppen- und Zusammenfassungssymbole sowie dem Rootsymbol ist allen Symbolen das jeweils symbolisierte Element assoziiert. Ein Zusammenfassungssymbol steht für mehrere Elemente, wohingegen Gruppensymbole im Strukturmodell keine Entsprechung haben. Bei Pfeilen gilt, daß nur Abhängigkeitspfeile nicht mit einer Beziehung assoziiert sind. Abhängig-

keiten werden erst im Zuge der Visualisierung gewonnen und sind daher nicht Bestandteil von Strukturmodellen. Die Teile eines Diagramms haben die Aufgabe, Eigenschaften der Gegebenheiten des Strukturmodells über die genannten Assoziationen abzufragen und in die Notation umzusetzen.

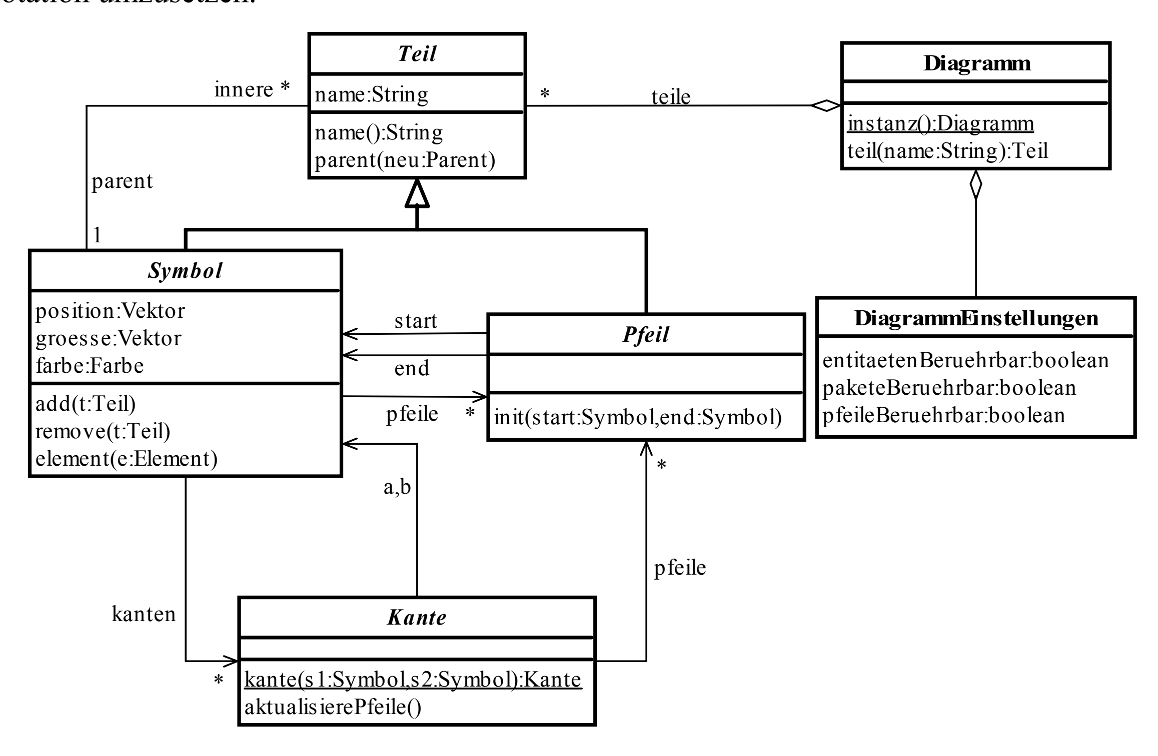


Abbildung 12.4: Interne Repräsentation von Diagrammen
Kursiv geschriebene Namen kennzeichnen in dieser und in den folgenden Abbildungen abstrakte Entitäten gegenüber konkreten Entitäten. Unterstreichungen markieren statische Methoden oder Variablen.

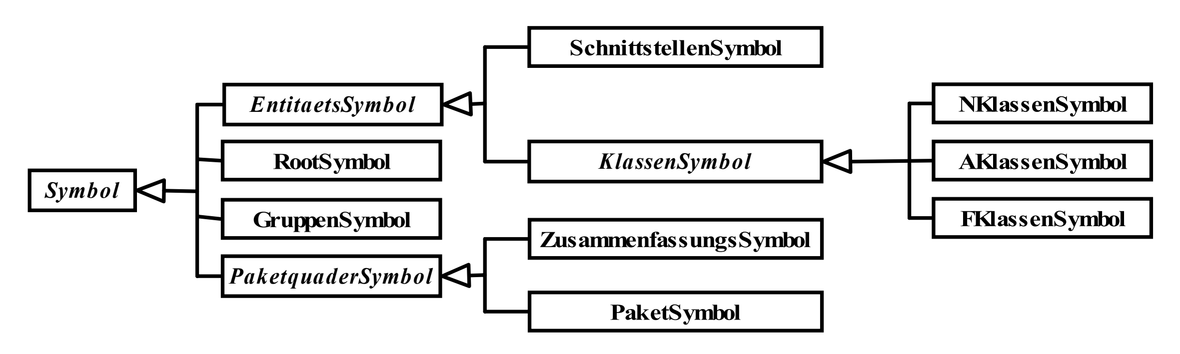


Abbildung 12.5: Spezialisierungen der Klasse „Symbol“

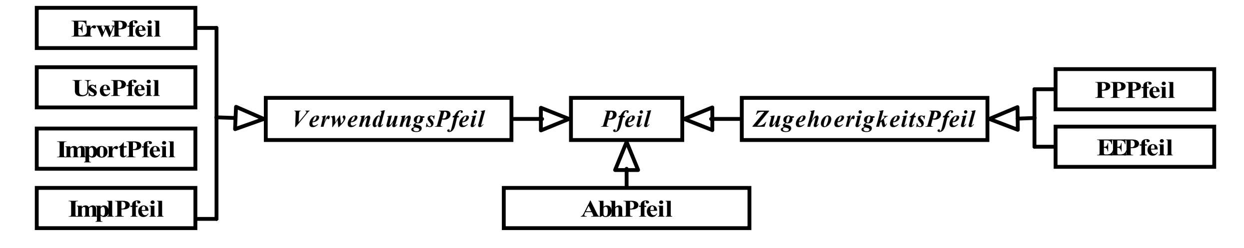


Abbildung 12.6: Spezialisierungen der Klasse „Pfeil“

Teile besitzen einen eindeutigen Namen. Dieser ist für Teile, die für Gegebenheiten aus dem Strukturmodell stehen, immer gleich dem Namen der entsprechenden Gegebenheit. Für Symbole ohne Entsprechung im Strukturmodell, wie z.B. für Gruppensymbole, wird der Name vom Benutzer festgelegt. Bei Abhängigkeitspfeilen geschieht die Namensvergabe wie

bei Beziehungen, wobei das Kürzel „Abh“ verwendet wird. Anhand des Namens kann vom Diagramm mittels der Methode *teil* das entsprechende Diagrammteil abgefragt werden.

Die Klasse *Symbol* bietet Methoden, mit denen u.a. Position, Größe und Farbe von Symbolen verändert werden können. Die Position eines Symbols wird dabei relativ zum Mittelpunkt des umschließenden Symbols gespeichert. Dies hat den Vorteil, daß bei einer Positionsänderung eines umschließenden Symbols, bei der innere Symbole ja ebenfalls bewegt werden müssen, die Positionsangaben dieser inneren Symbole trotzdem unverändert bleiben können.

Alle Pfeile zwischen zwei Symbolen werden unabhängig von ihrer Richtung zu einer Kante zusammengefaßt. Dieses Objekt ist für die optische Trennung bzw. Zusammenfassung der Pfeile verantwortlich (vgl. Abschnitt 12.5.5). Weiterhin erfolgt anhand der Kanten die automatische Ausrichtung (vgl. Abschnitt 12.7).

12.5.2 Initialisierung der internen Repräsentation

Die Initialisierung eines Diagramms wird durch eine eigene Klasse gesteuert. Diese heißt *DiagrammEANew*. Sie erbt von *DiagrammEA*, wo Gemeinsamkeiten verschiedener Eingabe-/Ausgabeoperationen, wie z.B. das Anfordern von Berechtigungen mittels *HTMLBrowser*, gebündelt sind.

Nachdem das zu visualisierende Strukturmodell über den Serialisierungsmechanismus von Java eingelesen wurde, erfolgt die weitere Initialisierung durch die Methode *initialisierung* von *DiagrammEANew* in fünf Schritten:

1) Für die Elemente des Strukturmodells werden entsprechende Symbole erzeugt.
2) Den Symbolen wird eine initiale Position zugewiesen.
3) Für die Beziehungen des Strukturmodells werden Pfeile erzeugt.
4) Pfeile für Abhängigkeiten werden erzeugt.
5) Das Diagramm wird graphisch realisiert.

Erzeugen von Symbolen. Um für die in einem Strukturmodell enthaltenen Elemente Symbole zu erzeugen, werden die Elemente rekursiv durchlaufen. Dies geschieht über die Klasse *SymbolErzeuger*, die eine Subklasse von *GegebenheitsErwVisitor* ist. Nur die Methode *visitElement* wurde redefiniert. Bei einem Aufruf für ein Element *e* geschieht folgendes:

- Eine Subklasse von *Symbol* wird instanziiert. Hierfür ist die Klasse *SymbolInstanziierer* verantwortlich. Diese ist wie *SymbolErzeuger* eine Subklasse von *GegebenheitsErw-Visitor*. Hier sind alle *visit*-Methoden der konkreten Subklassen von *Element* redefiniert, d.h. für *Paket* und *Schnittstelle* sowie für *N*-, *A*- und *FKlasse*. Die Redefinition besteht darin, eine Instanz der entsprechenden Subklasse von *Symbol* zu erzeugen. Sie lautet also bspw. für *NKlasse*:

  ```
  public Object visitNKlasse(NKlasse g) { return new NKlassenSymbol(); }
  ```

  Eine Instanz von *SymbolInstanziierer* wird der Methode *accept* des Elements *e* übergeben, wodurch der Aufruf der korrespondierenden *visit*-Methode erfolgt und die richtige *Symbol*-Subklasse instanziiert wird. Durch diese Nutzung des *Visitor*-Entwurfmusters wird ein umständliches *if-then-else*-Konstrukt der Form

  ```
  if( e instanceof NKlasse)     { return new NKlassenSymbol(); }
  else if(e instanceof AKlasse) { return new AKlassenSymbol(); }
  ...
  ```

  vermieden.

- Das erzeugte Symbol wird mit *e* verbunden. Hierfür ist die Methode *element* von *Symbol* zuständig. Die Eigenschaften von *e* werden erst später während der graphischen

Realisierung (s.u.) abgefragt.

- Für das erzeugte Symbol wird ein äußeres Symbol gesucht. Dies geschieht in der Methode *parentFuer* von *SymbolErzeuger*. Während der Initialisierung liefert diese Methode für Pakete stets die *RootSymbol*-Instanz als Parent. Entitätssymbole werden für gewöhnlich in das Symbol des jeweiligen Pakets geschachtelt. Eine Ausnahme bilden Symbole für innere Entitäten, deren Parent das Symbol der äußeren Entität ist.
- Das erzeugte Symbol wird dem gefundenen äußeren Symbol zugewiesen. Damit gehört das neue Symbol zum Diagramm.

Zuweisung einer initialen Position. Die Positionsangabe jedes Symbols wird mit einem zufälligen Wert initialisiert. Einzige Randbedingung ist es dabei, daß jedes Symbol innerhalb seines Parents positioniert wird. Verantwortlich für die Positionierung ist die Klasse *Random-Layout*. Nach Abschluß der Initialisierung kann die Positionierung vom Benutzern des Systems mittels der automatischen Ausrichtung verbessert werden.

Erzeugen von Pfeilen. Der nächste Schritt bei der Initialisierung besteht darin, für die Beziehungen des Strukturmodells Pfeile zu erzeugen. Dafür ist die Methode e*rzeugePfeile* der Klasse *PfeilErzeuger* verantwortlich. Hier werden alle im Strukturmodell gespeicherten Beziehungen durchlaufen. Für diese werden Pfeile instanziiert und initialisiert.

Um für eine Beziehung Instanzen der jeweils richtigen Subklasse von *Pfeil* zu erzeugen, wird ähnlich vorgegangen wie bei der Instanziierung von *Symbol*-Subklassen: Es gibt dazu eine Subklasse von *GegebenheitsErwVisitor* namens *PfeilInstanziierer* von der eine Instanz der Methode *accept* von *Beziehung* übergeben wird. Die Instanz führt dann die Instanziierung durch.

Nachdem ein Pfeil konstruiert wurde, gilt es diesen mit der dargestellten Beziehung zu verbinden und ihn zu initialisieren. Die Initialisierung eines Pfeils besteht darin, ihn in das Diagramm aufzunehmen und ihn den verbundenen Symbolen sowie der zwischen diesen bestehenden Kante zuzuordnen. Diese drei Dinge geschehen mittels der Methode *init*, die als Parameter Start- und Endsymbol eines Pfeils erwartet. Beide können anhand der Namen der in Beziehung stehenden Elemente ermittelt werden. Die Aufnahme eines Pfeils in das Diagramm geschieht dadurch, daß der Pfeil dem in der Schachtelungshierarchie tiefsten Symbol, welches die beiden verbundenen Symbole noch umschließt, als inneres Diagrammteil hinzugefügt wird.

Um die Kante für einen Pfeil finden zu können, verwaltet die Klasse *Kante* eine statische Variable mit einer Tabelle, durch die jedes Paar von Symbolen, zwischen denen mindestens ein Pfeil existiert, mit einer Kante verbunden ist. Die Pfeilrichtung ist dabei unerheblich. Diese Tabelle wird während einer Pfeilinitialisierung für ein aus dem Start- und Endsymbol bestehendes Paar abgefragt. Wird bei dieser Abfrage keine Kante gefunden, weil vorher noch kein Pfeil zwischen dem Symbolpaar existierte, so instanziiert die Abfragemethode eine neue Kante und nimmt sie in die Tabelle auf.

Erzeugen von Abhängigkeitspfeilen. Die Klasse *PfeilErzeuger* ist ebenfalls für das Erzeugen von Abhängigkeitspfeilen zuständig. Dies geschieht in der Methode *erzeugeAbhaengigkeits-pfeile* auf der Grundlage der im ersten Initialisierungsschritt berechneten Schachtelung von Symbolen und der im dritten Schritt entstandenen Pfeile.

Graphische Realisierung. Während der Realisierung werden von den Teilen des Diagramms Eigenschaften der dargestellten Gegebenheiten ausgelesen und graphisch umgesetzt. Dieser Vorgang basiert auf den Entitäten des Pakets *darstellung.graphik* und wird daher bei der Beschreibung dieses Paket im Abschnitt 12.8 erläutert.

12.5.3 Persistenz und Abgleich von Diagrammen

Um Strukturmodelle zu speichern, wird der Java-Serialisierungsmechanismus genutzt, der binäre Dateien erzeugt. Für Darstellungsdateien wurde ein anderer Weg gewählt. Hier werden Textdateien verwendet. Dies wurde am Anfang der Arbeit festgelegt, da zunächst geplant war, daß Veränderungen an Diagrammen allein über die Manipulation von Darstellungsdateien geschehen können. Dieser batchorientierte Ansatz erwies sich aber schnell als unbrauchbar, da während einer Veränderung deren Auswirkungen nicht direkt sichtbar werden und so sehr viel Vorstellungskraft bei der Diagrammgestaltung notwendig ist. Daher wurde eine graphische Benutzungsschnittstelle geschaffen. Trotzdem blieb es bei Textdateien, um einen Umstellungsaufwand zu vermeiden. Eine manuelle Veränderung der Darstellungsdateien ist aber nicht mehr empfehlenswert, da diese mittlerweile einen recht komplexen Aufbau besitzen.

Darstellungsdateien sind in Blöcke unterteilt. Jeder Block besteht aus mehreren Zeilen und endet mit zwei Nummernkreuzen (vgl. Dateiausriß 12.1). Jede Zeile enthält die Festlegung einer Eigenschaft. Im Paket *util* existieren Klassen wie *Block*, *BlockLeser* und *Block-Schreiber*, die eine Nutzung derartig aufgebauter Klassen erleichtern.

Bei Darstellungsdateien beschreibt außer den ersten drei Blöcken jeder Block ein Symbol. Andersartige Diagrammteile, wie z.B. Pfeile, werden derzeit nicht gespeichert. Die Reihenfolge der Blöcke wird durch die Schachtelungshierarchie der Symbole bestimmt: dem Block eines Symbols folgen die Blöcke für dessen innere Symbole.

Dateiausriß 12.1: [Auschnitt aus einer Darstellungsdatei]

```
ART=de.j3browser.darstellung.diagramm.PaketSymbol
Name=java.awt
Parent=Diagramm
Farbe=(0.0, 0.56078434, 0.5352941)
Groesse=(50.0, 30.0, 50.0)
Position=(391.44205, 15.75, 437.14288)
...
##
ART=de.j3browser.darstellung.diagramm.SchnittstellenSymbol
Name=java.awt.ItemSelectable
Parent=java.awt
Farbe=(0.0, 0.56078434, 0.8352941)
Groesse=(1.0, 1.0, 1.0)
Position=(-7.3690677, -5.8063946, 3.2898014)
...
##
```

Eine wichtige Symboleigenschaft ist die Art. Hier wird die Klasse angegeben durch die das Symbol repräsentiert wird. Diese Angabe wird dazu genutzt, die entsprechende Klasse beim Laden zu instanziieren. Dies geschieht in der statischen Methode *factory* von *Symbol*:

```
static Symbol factory(String art) {
  Class symClass = Class.forName(art);
  Constructor constructor = symClass.getConstructor(NOARGS_FORMAL);
  return (Symbol) constructor.newInstance(NOARGS_AKTUELL);
}
```

Während der Laufzeit einer Java-Software existiert für jede Klasse eine *Class*-Instanz, die sie repräsentiert. Diese Instanz wird für die angegebene Art ermittelt. Daraufhin wird in ihr ein

parameterloser Konstruktor gesucht, der bei allen Subklassen von *Symbol* vorhanden ist. Durch ihn entsteht eine neue Instanz einer konkreten Subklasse von *Symbol*.

Insgesamt wird das Laden durch die Klasse *DiagrammEALoad* gesteuert. Diese Subklasse von *DiagrammEA* ist auch für den Abgleich von Diagrammen zuständig. Eine Darstellungsdatei wird von ihr sequentiell gelesen. Der erste Block enthält einen Verweis auf die Datei des dem Diagramm zugrundeliegenden Strukturmodells. Die nächsten zwei Blöcke enthalten Angaben zu den im Einstellungsfenster festgelegten Ebenenfarben (siehe Abschnitt 10.8) und eine sogenannte Pfeilnegativliste (s.u.).

Für die weiteren Blöcke werden die entsprechenden Symbole mittels *factory* instanziiert. Als Bestandteil des Abgleichs wird geprüft, ob das symbolisierte Element im Strukturmodell vorhanden ist. Wenn nicht, dann wird das Symbol wieder verworfen. Blöcke für innere Symbole werden dann ebenfalls ignoriert. Zusätzlich wird der Name des verworfenen Symbols gespeichert, um den Benutzer später über Veränderungen informieren zu können. Existiert das zu symbolisierende Element bzw. benötigt ein Symbol ein solches gar nicht, was z.B. bei Gruppensymbolen der Fall ist, werden die weiteren Eigenschaften über die Methode *load* von *Symbol* gelesen, wobei durch das Lesen der Eigenschaft *Parent* die Aufnahme in das Diagramm erfolgt. Die verschiedenen Subklassen redefinieren *load*, um die jeweils nur für sie relevanten Eigenschaften zusätzlich zu lesen.

Über die in Darstellungsdateien gespeicherten Namen von Symbolen kann jedes Symbol mit dem gleichnamigen Element aus dem aktuellen Strukturmodell verbunden werden. Dabei bekommen Elemente, für die ein Symbol gefunden wird, eine Markierung. Nachdem alle Symbole eingeladen wurden, wird im Rahmen des Abgleichs geprüft, ob alle Elemente eine Markierung haben, d.h. symbolisiert werden. Die Namen der Elemente ohne Symbol werden gesammelt. Der Benutzer des Systems kann entscheiden, ob für diese Elemente neue Symbole erzeugt werden sollen. Dazu wurde von der Realisierung der Benutzungsoberfläche beim Initiieren des Ladevorgangs eine Instanz einer Implementierung der Schnittstelle *AbgleichAsker* übergeben. Deren Methode *frageAbgleich* bekommt die gesammelten Namen der nicht-symbolisierten Elemente sowie auch die der Symbole ohne Element übergeben. Die derzeitige Implementierung prüft lediglich, ob überhaupt Namen vorhanden sind und gibt daraufhin eine entsprechende Meldung aus, ggf. zusammen mit einer Rückfrage nach der Instanziierung neuer Symbole. Erweiterte Implementierungen könnten diese Namen z.B. als Listen ausgeben, um den Benutzer umfassender zu informieren.

Das Erzeugen neuer Symbole geschieht auf dieselbe Weise wie bei der Initialisierung eines Diagramms. Auch das weitere Vorgehen gleicht der Initialisierung: dem Erzeugen der Pfeile (inkl. der Abhängigkeitspfeile) schließt sich die graphische Realisierung an. Bei der Pfeilerzeugung mittels der Beziehungen des Strukturmodells muß aber eines beachtet werden: es können Beziehungen existieren, bei denen die in Beziehung stehenden Elemente nicht oder nicht direkt symbolisiert werden. Dies kann zwei Gründe haben. Zum einen kann eines der Elemente neu sein und der Benutzer hat auf das Erzeugen neuer Symbole verzichtet. In diesem Fall wird die Beziehung ignoriert. Zum zweiten kann die Beziehung ein Paket betreffen, das in einer Paketzusammenfassung enthalten ist. Für das Paket selbst existiert dann kein Symbol. Für eine solche Beziehung wird ein Pfeil erzeugt, der aber zum Symbol der Paketzusammenfassung umgeleitet wird. Davon ausgeschlossen sind *PP*-Beziehungen, da eine Paketzusammenfassung keine Sub- oder Superpakete hat.

Im Eigenschaftenfenster des J3Browser kann die Darstellung jedes Pfeils einzeln verhindert werden. Auch hier ist eine Persistenz der getroffenen Einstellungen sinnvoll. Pfeile werden aber in Darstellungsdateien nicht wie Symbole gespeichert. Statt dessen umfaßt jede Darstellungsdatei einen Block mit einer Pfeilnegativliste. Hier sind die Namen der Pfeile eines Diagramms aufgelistet, deren Darstellung zuvor verhindert wurde.

DiagrammEA besitzt neben den erwähnten Klassen *DiagrammEALoad* und *DiagrammEANew* noch zwei weitere Subklassen. Das Speichern von Diagrammen wird durch die Subklasse *DiagrammEASave* gesteuert. Die Klasse *DiagrammEAExport* ist für den VRML-Export von Diagrammen verantwortlich. Diese Klassen nutzen die Methoden *save* von *Symbol* und *export* von *Teil*, um Symbole zu speichern bzw. Diagrammteile zu exportieren.

12.5.4 Sichtbarkeit von Diagrammteilen

Die Sichtbarkeit eines Diagrammteils innerhalb eines Diagramms wird durch verschiedene Faktoren bestimmt. So kann der Benutzer die Darstellung jedes Symbols und jedes Pfeils einzeln unterbinden. Zudem kann er Entitätssymbole und Pfeile filtern, sowie die Isolierung von Symbolen fordern. Durch die Isolierung sowie der unterbundenen Darstellung von Symbolen können auch zugehörige Pfeile unsichtbar werden. All dies wird durch die Methode *effektiveSichtbarkeit* von *Teil* modelliert. Diese liefert als Ergebnis eine Konjunktion der verschiedenen die Sichtbarkeit bestimmenden Faktoren. Da bei den unterschiedlichen Arten von Diagrammteilen verschiedene Faktoren relevant sind, wird die Methode von Subklassen redefiniert. Die effektive Sichtbarkeit eines Pfeils bestimmt sich bspw. wie folgt:

```
public boolean effektiveSichtbarkeit() {
  boolean neueEffektiveSichtbarkeit =
    super.effektiveSichtbarkeit() && verbindungGueltig() &&
    !zuInneren(start,end) && !zuInneren(end,start) &&
    startEndOk() && inFiltermenge();
  ...
}
```

Damit ein Pfeil sichtbar ist, muß zunächst die Implementierung aus *Teil* die Sichtbarkeit bestätigen. Sie tut dies, wenn die Darstellung des einzelnen Teils nicht manuell durch den Benutzer unterbunden wurde. Weiterhin muß die Verbindungslinie des Pfeils gültig sein, d.h. Start- und Endsymbol dürfen sich nicht auf derselben Position befinden. Zudem sind Pfeile unsichtbar, die von äußeren zu inneren oder von inneren zu äußeren Symbolen verlaufen. In der Methode *startEndOk* wird geprüft, ob sowohl Start- als auch Endsymbol sichtbar sind und ob keine Isolierungen von Symbolen der Sichtbarkeit des Pfeils im Wege stehen. Schließlich prüft *inFiltermenge*, ob nicht ein gesetzter Filter die Darstellung des Pfeils verhindert. Die Ausgestaltung des Filters wird durch eine mit dem Diagramm assoziierte Instanz der Klasse *FilterSpezifikation* bestimmt (vgl. Abbildung 12.7),. Eine Filterspezifikation besitzt für jede Pfeilart zwei boolsche Einträge, die bestimmen, ob lokale Pfeile, d.h. Pfeile, bei denen die verbundenen Symbole denselben Parent besitzen, und/oder globale Pfeile dieser Art angezeigt werden. Für lokale und globale Benutzungspfeile werden zusätzlich minimale Zugriffsmodi festgelegt. Gleiches gilt für Entitätssymbole.

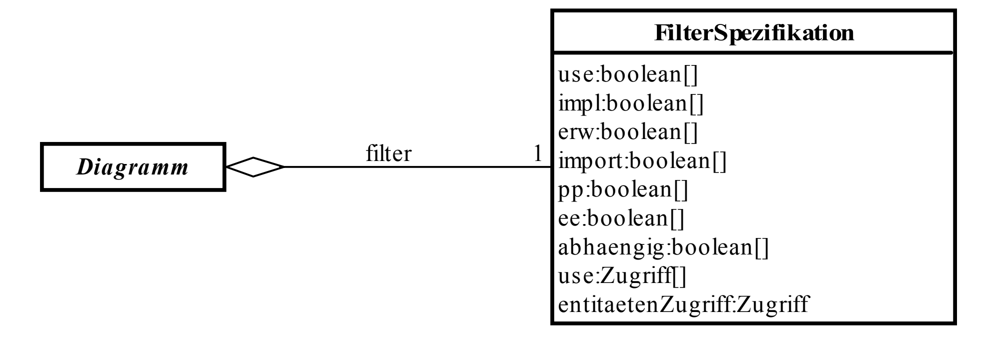


Abbildung 12.7: Klasse „FilterSpezifikation“

Will man dem J3Browser weitere Visualisierungstechniken hinzufügen, welche die

Sichtbarkeit von Teilen bestimmen, wie bspw. solche, die mit dem Rechnen mit Relationen bei ArchView vergleichbar sind, kann dies zweckmäßig durch eine erweiterte Berechnung der effektiven Sichtbarkeit geschehen.

12.5.5 Optische Zusammenfassung und Trennung von Pfeilen

Wenn Pfeile ihre Sichtbarkeit ändern, kann es gleichzeitig erforderlich sein, verschiedene Pfeile zwischen zwei Symbolen optisch zu trennen oder auch zusammenzufassen. Gegenläufige Pfeile derselben Art werden zusammengefaßt, so daß z.B. aus zwei symmetrischen Pfeilen für Benutzungen ein Pfeil mit zwei Spitzen entsteht. Pfeile unterschiedlicher Art müssen voneinander getrennt werden, damit ein Betrachter sie auseinanderhalten kann. Für beide Vorgänge ist die Klasse *Kante* verantwortlich. Alle Pfeile zwischen zwei Symbolen werden richtungsunabhängig einer Kante zugeordnet. Diese Pfeile informieren die Kante über Änderungen ihrer Sichtbarkeit. Die Kante führt Zusammenfassung und Trennung durch, indem die sogenannten Ansatzpunkte eines Pfeils verändert werden. Diese bestimmen, wo ein Pfeil die beiden verbundenen Symbole berührt.

Zunächst ist die jeweilige Subklasse von *Symbol* dafür zuständig, den Ansatzpunkt eines Pfeils zu bestimmen. Dies geschieht in Abhängigkeit von der Form des Symbols und nach Möglichkeit so, daß der Ansatzpunkt auf der Symboloberfläche liegt, da dies optisch ansprechender ist. Erkennt nun eine Kante, daß mehr als einer ihrer Pfeile sichtbar ist, werden deren Enden in einem Kreis um den vom Symbol berechneten Ansatzpunkt angeordnet. Von der Kante wird dabei für jeden Pfeil ein Winkel auf dem Kreisbogen bestimmt. Gleichartige Pfeile verschiedener Richtung wird der gleiche Winkel zugeordnet, so daß sie zusammengefaßt werden, wohingegen verschiedenartige Pfeile gleichmäßig über den Kreisbogen verteilt werden. Bei zwei zusammengefaßten Pfeilen wird also nicht eine Verbindungslinien gemeinsam genutzt, sondern es werden beide Linien übereinander dargestellt. Diese Lösung wurde wegen ihrer leichteren Umsetzbarkeit gewählt. Allerdings sind so in der dargestellten Szene ein wenig mehr graphische Objekte enthalten als unbedingt notwendig sind. Um diesen Umstand zu begegnen, müßte für jeden Pfeil die Sichtbarkeit seiner Pfeilspitze getrennt von der der Verbindungslinie gewechselt werden können. Dies ist derzeit nicht der Fall.

12.5.6 Selektion und Berührung

Die Selektion von Symbolen wird über die vier Entitäten *Selektion*, *Selektionserweiterung, Selektion.Listener* und *SelektionsHighlighter* gehandhabt. Die Singleton-Klasse *Selektion* kapselt die Menge der ausgewählten Symbole. Die Funktionen zur Selektionserweiterung aus dem Abschnitt 10.4 werden durch *Selektionserweiterung* realisiert. An Veränderungen der Selektion interessierte Klassen müssen die innere Schnittstelle *Listener* von *Selektion* implementieren. Instanzen dieser Klassen können sich bei der Selektion registrieren lassen und werden dann durch den Aufruf von Methoden der Schnittstelle über Veränderungen informiert. Die Klasse *SelektionsHighlighter* ist eine solche interessierte Klasse. Sie sorgt für die optische Hervorhebung selektierter Symbole.

Auf ähnliche Weise wie bei der Selektion wird bei Berührungen eines Symbols oder Pfeils verfahren. Um diese zu handhaben, existiert die Singleton-Klasse *BeruehrungsManager*, welche ebenfalls eine innere Schnittstelle *Listener* umfaßt. Durch das Implementieren dieser Schnittstelle sowie eine Registrierung bei der *BeruehrungsManager*-Instanz kann eine Benachrichtigung über Berührungen gefordert werden.

12.5.7 DOIManager, DokuManager und EinblendungsManager

Die drei Singleton-Klassen *DokuManager*, *DOIManager* und *EinblendungsManager* sind genau wie *SelektionsHighlighter* an Veränderungen der Selektion interessiert. Mit ihnen wird die Anzeige von Dokumentation, die Degree-of-Interest-Darstellung für Pfeile und die

temporäre Einblendung von Symbolen umgesetzt.

Informationen über die Arbeitsweisen der einzelnen Manager können der Javadoc-Dokumentation zum J3Browser entnommen werden (vgl. Anhang A). Als ein Beispiel soll hier auf den DOIManager eingegangen werden. Anhand einer *FilterSpezifikation*-Instanz wird festgelegt, für welche Arten von Pfeilen dieser aktiv werden soll. Das Verändern dieser Instanz führt genauso wie ein Wechsel der Hauptselektion zu einer Aktualisierung der Darstellung. Diese geschieht in zwei Schritten durch die Methode *brennpunkt*: zunächst wird für jeden Pfeil dessen Entfernung von der Hauptselektion berechnet. Im zweiten Schritt werden aus diesen Entfernungen die DOI-Werte der Pfeile aktiver Arten abgeleitet. Die Werte geben direkt den Grad der Transparenz der Pfeile an.

Während der Entfernungsberechnung werden zwei Tabellen aufgebaut: eine, die den Namen von Pfeilen auf Entfernungswerte abbildet und eine, die gleiches für die Namen von Symbolen tut. Während die erste Tabelle das Ergebnis der Entfernungsberechnung bildet, wird die zweite Tabelle ausschließlich während der Berechnung genutzt. Da der DOIManager bei jedem Selektionswechsel aktiv wird, muß dessen Arbeit performant geschehen. Dazu umfassen die Tabellen nicht alle Symbole und Pfeile des Diagramms, sondern reichen nur bis zu einer maximalen Entfernung. Weiterhin sind sie als Hash-Tabellen realisiert. Für beide Tabellen existiert jeweils eine Zugriffsmethode *get*, die für ein Symbol oder einen Pfeil die Entfernung aus der Tabellen liefert, oder, wenn kein entsprechender Tabelleneintrag existiert, einen Maximalwert liefert. Analog gibt es zwei *set*-Methoden, die die in der jeweiligen Tabelle gespeicherte Entfernung setzen.

Anfänglich befindet sich in der Tabelle für die Symbolenfernungen nur ein Eintrag, der für das hauptselektierte Symbol die Entfernung 0 festlegt. Es wird die Methode *entfernungen-Berechnen* aufgerufen, die eine Liste von zu bearbeitenden Symbolen erwartet (TODO-Liste). Beim ersten Aufruf wird eine nur aus dem hauptselektiertem Symbol bestehende Liste übergeben. Die Methode arbeitet wie folgt:

```
TODO‘ = {}
forall s ∈ TODO
    sentfernung = get(s);
    if sentfernung <= maximale Entfernung
        forall effektiv sichtbaren Pfeile p von s in passender Richtung
            if get(p) >= sentfernung
                set(p,sentfernung);
                neueSymbolEntfernung = sentfernung+pfeillänge(p);
                if neueSymbolEntfernung < get(pfeil.start)
                    set(pfeil.start, neueSymbolEntfernung)
                    TODO‘ = TODO‘ ∪ { pfeil.start }
                if neueSymbolEntfernung < get(pfeil.end)
                    set(pfeil.end, neueSymbolEntfernung)
                    TODO‘ = TODO‘ ∪ { pfeil.end }
if TODO‘ <> {}
    entfernungenBerechnen(TODO‘)
```

Abbildung 12.8 auf folgender Seite zeigt ein Beispiel für den Ablauf einer Berechnung. Im Beispiel gilt nur die Pfeilrichtung als passende Richtung. Der Systembenutzer kann zusätzlich oder alternativ die Gegenrichtung zur passenden Richtung machen (vgl. Abschnitt 10.5.2). Vor dem ersten Schritt besteht die ganz links abgebildete Situation: nur das als hauptselektiert geltende Symbol A besitzt eine Entfernungsangabe und es ist lediglich A zu bearbeiten. Bei dieser Bearbeitung werden die Pfeile zu B und E sowie die beiden Symbole selbst mit

Entfernungsangaben versehen. Eine Pfeilentfernung ist immer gleich der kleineren Entfernungen der beiden verbundenen Symbole. Symbolentfernungen werden mit Hilfe der Methode *pfeillänge* ermittelt. Diese liefert eine rechnerische Länge des übergebenen Pfeils. Derzeit wird hier immer 1 geliefert, erweiterte Implementierungen könnten aber z.B. die Art des Pfeil berücksichtigen, so daß etwa Erweiterungspfeile stärker verbindend wirken als Benutzungspfeile. Ergibt sich über den betrachteten Pfeil ein kürzerer Weg zum verbundenen Symbol, so wird dessen Entfernung aktualisiert und das Symbol in eine neue TODO-Liste aufgenommen. Solange eine nicht leere neue TODO-Liste vorliegt, wird für diese am Ende eines Schrittes *entfernungenBerechnen* aufgerufen. Abbildung 12.8 zeigt, von links nach rechts, die weiteren Situationen vor dem ersten, zweiten und dritten Schritt sowie nach dem dritten Schritt.

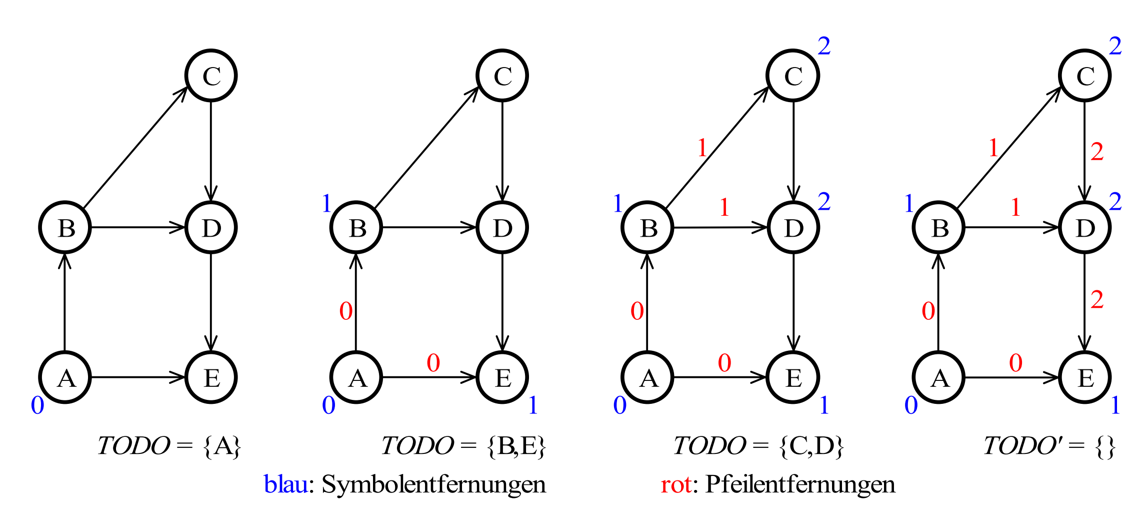


Abbildung 12.8: Ablauf der Entfernungsberechnung

Die Entfernungsberechnung wird nur bis zu einer maximalen Entfernung durchgeführt, allen nicht betrachteten Pfeilen wird durch die *get*-Methode ebenfalls der Entfernungsmaximalwert zugeordnet. Gleichzeitig wird bei der Berechnung nicht unterschieden, ob ein betrachteter Pfeil von einer Art ist, für die die DOI-Anzeige aktiviert ist. Dies erfolgt erst bei der Transparenzberechnung. So wird es möglich, daß bspw. Pfeile für Erweiterungsbeziehungen verbindend wirken, sie selbst aber nicht der DOI-Anzeige unterliegen.

Nachdem bei einem Selektionswechsel alle neuen Pfeilentfernungen berechnet wurden, müssen die Transparenzwerte der Pfeile verändert werden. Wann immer die Einstellung der aktiven Arten gewechselt wird, werden die Transparenzwerte für Pfeile einer aktiven Art auf die im Filterfenster angegebene Transparenz bei maximaler Entfernung gesetzt. Gleichzeitig werden bei einem solchen Einstellungswechsel Pfeile nicht-aktiver Arten opaque gemacht.

Beim ersten Selektionswechsel nach einer Einstellungsänderung reicht es zunächst aus, die Transparenz für nähere Pfeile zu wechseln, d.h. nur die Pfeile der Pfeilentfernungstabelle müssen berücksichtigt werden. Bei nachfolgenden Selektionswechseln müssen aber Pfeile, deren Transparenz zuvor verändert wurde, die aber nicht mehr in der neuen Pfeilentfernungstabelle enthalten sind, wieder die maximale Transparenz erhalten. Dazu ist es notwendig, daß nach einem Selektionswechsel auch die vormalig berechnete Pfeilentfernungstabelle noch zur Verfügung steht.

Die Transparenz der Pfeile wird durch die Methode *transparenzSetzen* des DOIManagers in zwei Schritten gesetzt. Zunächst werden alle Einträge der neuen Entfernungstabelle durchlaufen, dann die der alten Tabelle. Im ersten Schritt wird für jeden Eintrag der entsprechende Pfeil gesucht. Nur wenn dieser von einer aktiven Art ist, wird aus der Pfeilentfernung ein Transparenzwert abgeleitet und gesetzt. Die Ableitung geschieht derzeit durch lineare

Interpolation aus den im Filterfenster angegeben Transparenzwerten für nahe und maximal entfernte Pfeile.

Beim Durchlauf durch die alte Pfeilentfernungstabelle wird für jeden Eintrag geprüft, ob ein entsprechender Eintrag auch in der neuen Tabelle enthalten ist. Durch die Realisierung als Hash-Tabellen kann dies verhältnismäßig schnell festgestellt werden. Ist in der neuen Tabelle kein Eintrag vorhanden, so wird die Transparenz des entsprechenden Pfeils wieder auf den maximalen Wert gesetzt. Nach diesem Durchlauf wurde allen Pfeilen ein korrekter Transparenzwert zugeordnet und die Behandlung des Selektionswechsels durch den DOIManagers ist abgeschlossen.

12.5.8 Gruppierung von Symbolen

Symbole können zu Gruppen zusammengefaßt werden, die ikonifiziert werden können, wobei für ikonifizierte Gruppen Abhängigkeiten darzustellen sind. Ein Gruppe wird konstruiert, indem die Klasse *GruppenSymbol* instanziiert wird. Diese sowohl für Gruppensymbole in Zylinder- als auch in Kegelformen genutzte Klasse bietet die Methode *selektionGruppierbar*, mit der geprüft werden kann, ob die aktuell selektierten Symbole eine Gruppe bilden können. Eine Menge von Symbolen kann gruppiert werden, wenn alle enthaltenen Symbol denselben Parent besitzen, kein Symbol ein Gruppensymbol ist und kein Symbol bereits zu einer anderen Gruppe gehört. Nach der Prüfung wird mit *gruppiereSelektion* die Gruppierung durchgeführt. Dazu merkt sich das Gruppensymbol die gruppierten Symbole. Zudem erhalten die gruppierten Symbole einen Verweis auf das Gruppensymbol. Am Ende dieses durch die Klasse *GruppierungsFenster* des *gui*-Paketes gesteuerten Vorgangs wird das Gruppensymbol in das Diagramm aufgenommen.

Da neue Gruppen nicht ikonifiziert sind, dürfen neue Gruppensymbole zunächst noch nicht sichtbar sein. Deswegen ist die Berechnung der effektiven Sichtbarkeit von Gruppensymbolen um einen Test auf Ikonifizierung erweitert. Wird eine Ikonifizierung mittels der Methode *ikonifiziere* gefordert, so wird das entsprechende Gruppensymbol zum Parent der gruppierten Symbole gemacht. An dieser Stelle könnten Position und Größe der gruppierten Symbole so verändert werden, daß sie innerhalb des Gruppensymbols erscheinen. Derzeitig werden sie aber einfach unsichtbar gemacht. Dazu umfaßt die Berechnung der effektiven Sichtbarkeit von Symbolen auch ein Prüfen, ob das Symbol Bestandteil einer ikonifizierten Gruppe ist. Das Gruppensymbol selbst wird in der Mitte aller gruppierten Symbole plaziert.

Durch die Methode *deikonifiziere* werden die gruppierten Symbole wieder ihrem ursprünglichem Parent zugeordnet. Wie auch bei *ikonfiiziere* wird eine Neuberechnung der Abhängigkeiten initiieren.

Die Singleton-Klasse *GruppenHervorheber* hebt alle zu einer Gruppe gehörenden Symbole hervor, wenn eines der Symbol berührt wird. Dazu implementiert die Klasse *Beruehrungs-Manager.Listener*. Zur Hervorhebung wird die Farbe jedes Symbols etwas aufgehellt. Diese Hervorhebung wird wieder rückgängig gemacht, wenn die Berührung endet.

12.5.9 Manipulationen von Diagrammteilen

Wird an einem Diagrammteil eine Manipulation vorgenommen, sind oft weitere Aktionen erforderlich. So erzwingt die Veränderung einer Symbolposition das Aktualisieren des Verlaufs der mit dem Symbol verbundenen Pfeile. Ein weiteres Beispiel ist das Ändern der Darstellungsform eines Symbols. Wird diese z.B. anhand des Menüs von *einsehbar* auf *uneinsehbar* verändert, so muß auch die Symbolleiste aktualisiert werden, da dort die Darstellungsform des hauptselektierten Symbols angezeigt wird.

Um solche Aktualisierungen zum richtigen Zeitpunkt durchzuführen, wird hauptsächlich wieder das Entwurfsmuster *Observer* eingesetzt. Klassen wie *Symbolleiste* implementieren

die Schnittstelle *TeilListener*. Instanzen können sich dann beim interessierenden Teil registrieren lassen. Daraufhin werden sie über den Aufrufe einer Methode *updateTeil* über Veränderungen der Eigenschaften des Teils informiert. Allein zur Aktualisierung der mit einem Symbol verbundenen Pfeile werden vom Symbol direkt entsprechende Methoden von *Pfeil* aufgerufen. Würde sich jeder Pfeil bei den verbundenen Symbolen als Listener eintragen, so würde viel Speicherplatz verschwendet, da Symbole bereits über die Assoziation *pfeile* (vgl. Abbildung 12.4 auf Seite 96) einen Verweis auf ihre Pfeile enthalten.

12.6 Paket „darstellung.diagramm.anordnung"

12.6.1 Modellierung von Anordnungen

Dieses Paket realisiert die Visualisierungstechniken Baum, Cone Tree und Kegel, die unter dem Begriff der Anordnung zusammengefaßt werden. Entsprechend benannte Klassen sind für die Realisierung der Visualisierungstechniken verantwortlich (vgl. Abbildung 12.9).

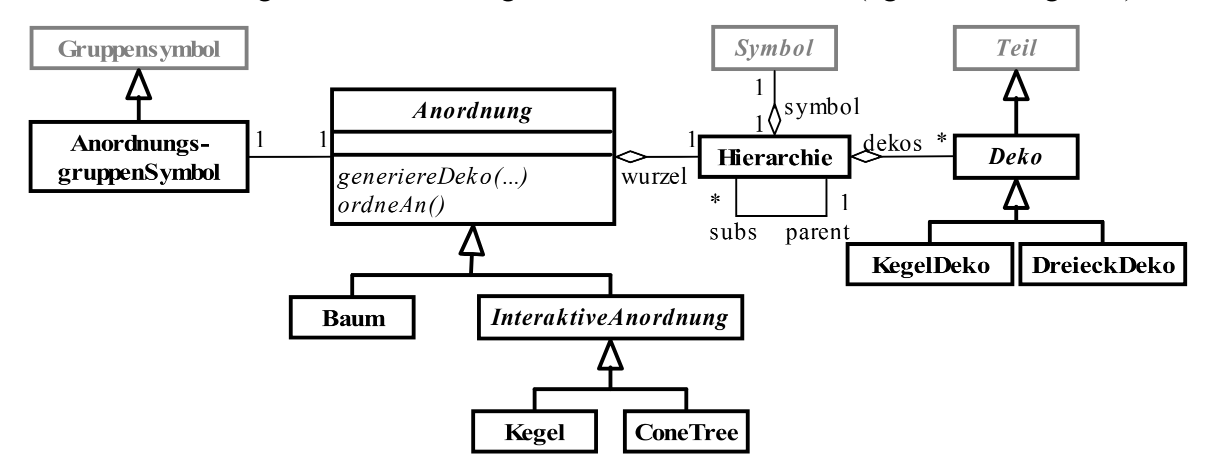


Abbildung 12.9: Modellierung von Anordnungen (Paketfremde Entitäten in grau)

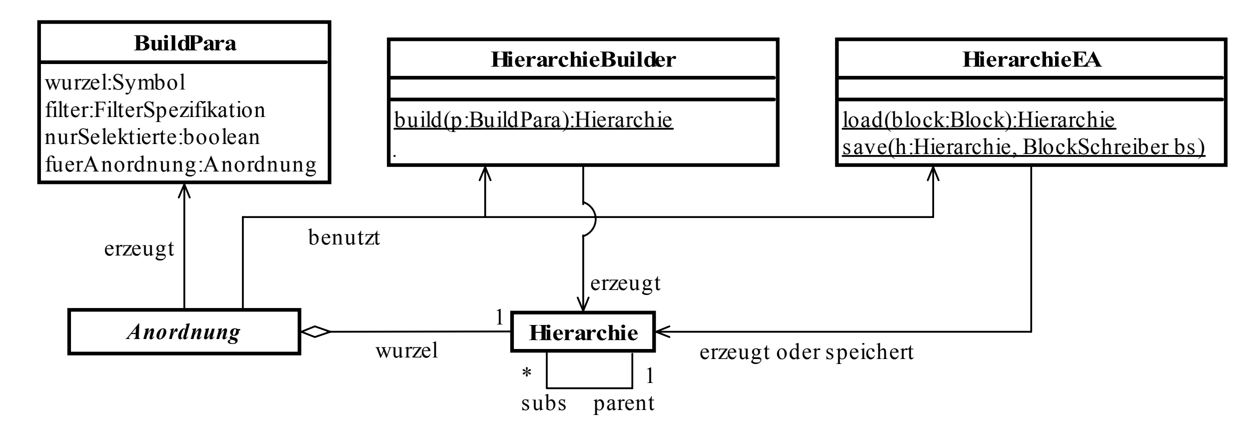


Abbildung 12.10: Erzeugen und Laden von Hierarchien

Anordnungen stellen aus Symbolen bestehende baumförmige Hierarchien dar. Systembenutzer spezifizieren eine darzustellende Hierarchie, indem sie ein Wurzelsymbol selektieren und die Arten von Pfeilen auswählen, die bei der Hierarchiebildung zu berücksichtigen sind. Aus diesen Angaben wird mit Hilfe der Klasse *HierarchieBuilder* (vgl. Abbildung 12.10) eine baumförmige Objektstruktur gewonnen, welche die darzustellende Hierarchie repräsentiert. Sie umfaßt für jedes in der Anordnung enthaltene Symbol eine Instanz von *Hierarchie*. Die *Hierarchie*-Instanz jedes Symbols ist über die Assoziation *subs* mit den *Hierarchie*-Instanzen der in der Anordnung direkt untergeordneten Symbole verbunden. Mit *parent* existieren auch Verweise in Gegenrichtung. Durch diese gesonderte Repräsentation ist die Arbeit der

Subklassen von *Anordnung* unabhängig von der Art der dargestellten Beziehung. Es ist somit nicht notwendig, eigene Spezialisierungen für Vererbungshierarchien, für Zugehörigkeiten usw. zu bilden. Es können sogar mehrere Beziehungsarten gleichzeitig behandelt werden, ohne daß dies in Subklassen von *Anordnung* berücksichtigt werden muß. Zudem werden manuelle Veränderungen in Hierarchien, wie insbesondere das Vertauschen von Teilbäumen, möglich.

Ein wichtiger Bestandteil von Cone Trees sind die semitransparenten Kegel. Diese werden hier als Dekorationen durch die Klasse *Deko* modelliert, bei der es sich um eine weitere Spezialisierung der Klasse *Teil* des Pakets *diagramm* handelt. Da es neben kegelförmigen Dekorationen für die anderen Arten von Anordnungen auch dreieckige Dekorationen gibt, existieren die zwei Spezialisierungen *KegelDeko* und *DreieckDeko*. Die Dekorationen verbinden immer das Symbol einer *Hierarchie*-Instanz mit den direkt untergeordneten Symbolen. Sie werden daher mit *Hierarchie*-Instanzen assoziiert. Die Farbe einer Dekoration ergibt sich aus der Tiefe des jeweiligen Symbols in der Hierarchie. Die Farbvergabe wird durch die Singleton-Klasse *Anordnungsfarben* realisiert. Sie kann über das Einstellungsfenster vom Benutzer verändert werden.

Wird eine Anordnung erzeugt, so müssen Symbole angeordnet und Dekorationen generiert werden. Hierfür sind die abstrakten Methoden *ordneAn* und *generiereDeko* verantwortlich, die durch die Subklassen implementiert werden. Je nach Subklasse werden die Symbole in der Form eines Baum, eines Cone Trees oder eines Kegels positioniert. Dabei werden sie automatisch gruppiert. Die so entstehenden Gruppen werden durch Anordnungsgruppensymbole symbolisiert. Die entsprechende Klasse ist eine Spezialisierung der für allgemeine Gruppen verwendeten Klasse *GruppenSymbol* aus dem Paket *diagramm*.

Das persistente Speichern einer Anordnung erfolgt zusammen mit der jeweiligen Hierarchie innerhalb einer Darstellungsdatei im Block des Anordnungsgruppensymbols. Für das Speichern und Laden von Hierarchien ist die Klasse *HierarchieEA* zuständig, wobei beim Laden ein Abgleich mit Änderungen im Strukturmodell notwendig werden kann. Dieser ist in der derzeitigen Implementierung recht einfach gehalten. Lediglich das Fehlen von Elementen in Strukturmodellen wird berücksichtigt, auf ein Hinzukommen neuer Elemente, die in eine bestehende Anordnung passen würden, muß der Benutzer eigenständig durch einen erneuten Aufbau der Hierarchie reagieren (vgl. Option *Neue Hierarchie* des im Abschnitt 10.6.4 beschriebenen Fensters für Anordnungen). Bei fehlenden Elementen gilt: ist ein Element, dessen Symbol Bestandteil einer Hierarchie ist, nicht mehr im Strukturmodell enthalten, so wird die entsprechende *Hierarchie*-Instanz aus der Hierarchierepräsentation der Anordnung entfernt. Dies bedingt, daß auch untergeordnete Symbole aus der Anordnung gelöst werden müssen, da es in Anordnungen keine Lücken geben kann.

12.6.2 Interaktive Drehung

Bei Cone Trees und Kegeln gilt es deren interaktive Drehung zu verwirklichen. Diese sorgt dafür, daß ein hauptselektiertes Symbol möglichst direkt vor der Betrachterposition erscheint. Hierfür ist die gemeinsame Superklasse *InteraktiveAnordnung* verantwortlich, die dazu die Schnittstelle *SelektionsListener* implementiert (vgl. Abbildung 12.11 auf nachfolgender Seite). Dies zeigt, wie das Observer-Muster genutzt werden kann, um mit der Selektion weitere Aktionen – z.B. zusätzliche Visualisierungstechniken – zu verknüpfen, ohne daß die Klasse *Selektion* dazu geändert werden muß.

Die Klasse *InteraktiveAnordnung* stützt sich bei der Rotation von Anordnungen auf die Klasse *Drehmaschine*. Während erste zusammen mit ihren Subklassen für die Berechnung der Drehung verantwortlich ist, ist es die Aufgabe der Drehmaschine, die Drehung sichtbar durchzuführen. Beim Starten der Drehmaschine übergibt eine interaktive Anordnung der

Maschine eine Spezifikation der durchzuführenden Drehung. Die Spezifikation wird in der Methode *drehSpezifikation* berechnet. Da sich die Drehung von Cone Trees und Kegeln unterscheidet, wird die Methode erst in den Subklassen implementiert. Eine Drehspezifikation besteht aus einer Menge von *DrehSpezElement*-Instanzen. Jedes dieser Drehspezifikationselemente umfaßt einen Verweis auf eine *Hierarchie*-Instanz der Anordnung und eine Angabe, die besagt, um welchen Winkel die durch die *Hierarchie*-Instanz modellierte Teilhierarchie um ihre Wurzelsymbol gedreht werden soll.

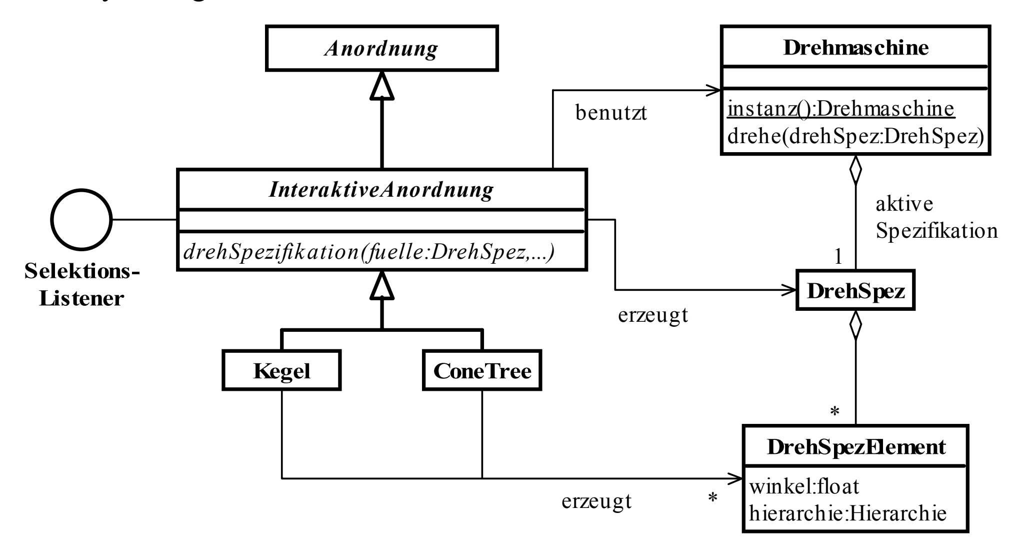


Abbildung 12.11: Interaktive Drehung von Anordnungen

Ein Pfad durch einen Cone Tree, der bei der Wurzel beginnt und mit dem Symbol endet, welches dem hauptselektierten Symbol direkt übergeordnet ist, wird nachfolgend als *Selektionspfad* des Cone Trees bezeichnet. Bei der Drehungsberechnung wird für jedes Symbol auf einem solchen Pfad ein Drehspezifikationselement erzeugt, d.h. genauer für die *Hierarchie*-Instanzen dieser Symbole. Bei dem in Abbildung 12.12 gezeigten Beispiel würde für *Hierarchie*-Instanzen der Symbole s_1, s_2 und s_3 je ein Spezifikationselement erzeugt.

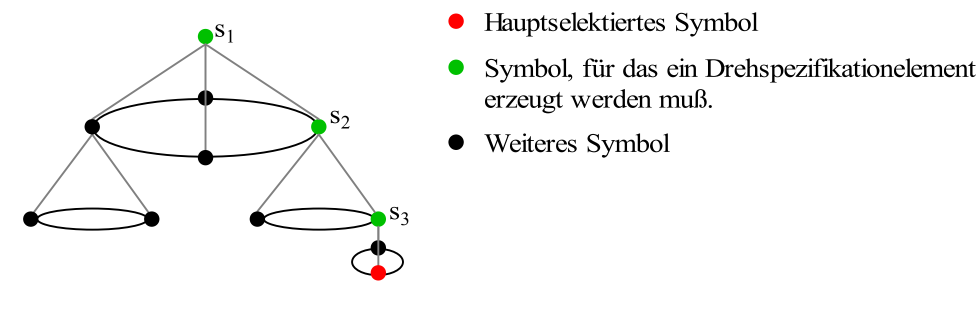


Abbildung 12.12: Benötigte Drehspezifikationselemente bei Cone Trees

Der Winkel für die Drehung um ein Symbol s_i befindet sich zwischen der Strecke von s_i zur Betrachterposition und der Strecke von s_i zum nächsten Symbol auf dem Selektionspfad bzw., wenn s_i das letzte Symbol des Pfads ist, von s_i zum hauptselektierten Symbol (vgl. Abbildung 12.13 auf nachfolgender Seite). Da bei Anordnungen ausschließlich Drehungen um Achsen möglich sind, die parallel der Y-Achse verlaufen, brauchen bei der Berechnung der Winkel die Y-Koordinaten nicht berücksichtigt werden. Ist so der erste Winkel berechnet, wird sozusagen „virtuell“ eine Drehung des ganzen Cone Trees um die Wurzel durchgeführt. Dies bedeutet, daß die neuen Koordinaten in die weiteren Berechnungen eingehen, eine Änderung der Symbolposition in der Darstellung aber zunächst nicht erfolgt. An diese virtuelle Drehung schließt sich die Berechnung des Winkels für das nächste Symbol auf dem Selektionspfad inklusive einer weiteren virtuellen Drehung an usw. Abbildung 12.13 zeigt den Vorgang für

obiges Beispiel. Dabei ist zu beachten, daß in diesem konkreten Fall keine Drehung um s_2 erforderlich ist.

Bei Kegeln ist die Berechnung einfacher. Hier ist nur ein Spezifikationselement erforderlich. Der gesamte Kegel wird um das Wurzelsymbol gedreht und zwar um einen Winkel, der sich zwischen der Strecke vom Wurzelsymbol zur Betrachterposition und der Strecke vom Wurzelsymbol zum hauptselektierten Symbol befindet.

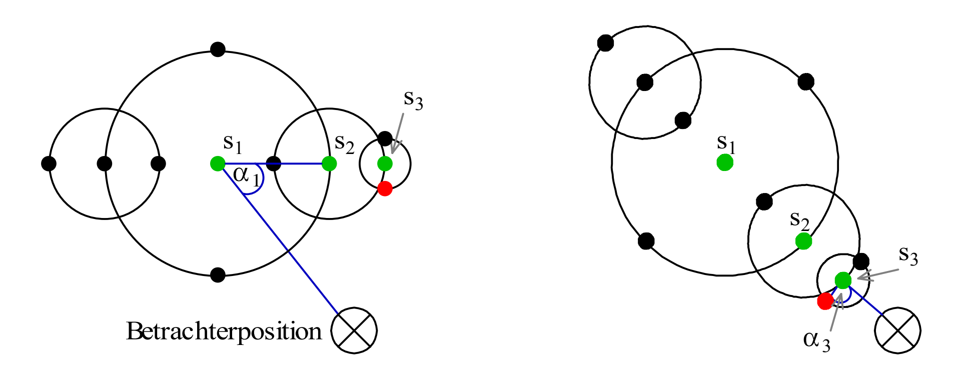


Abbildung 12.13: Berechnung der Drehspezifikationselemente bei Cone Trees

Die sichtbare Drehung erfolgt durch die Drehmaschine. Diese arbeitet in einem eigenen Thread. Solange keine Drehung durchgeführt wird, ist der Thread gestoppt (zur Verwendung von Threads in Java vgl. z.B. [Lea97]). Er nimmt erst mit der Übergabe einer Drehspezifikation mittels *drehe* seine Arbeit auf. Die Drehung erfolgt für Cone Trees und Kegel auf die gleiche Weise in mehreren Schritten. Bei jedem Schritt wird für jedes Spezifikationselement die spezifizierte Drehung für einen Bruchteil des angegebenen Winkels ausgeführt. Ist die Drehspezifikation komplett durchlaufen, wird der Thread für einige Millisekunden gestoppt, wobei die graphische Darstellung aktualisiert wird. Dann beginnt ein neuer Schritt. Aus der entlang des Selektionspfad sequentiell berechneten Drehspezifikation entstehen somit simultane Rotationen um mehrere Achsen, was optisch ansprechender ist. Die Drehmaschine beendet ihre Arbeit, wenn alle spezifizierten Drehungen vollständig abgearbeitet sind.

12.7 Paket „darstellung.diagramm.ausrichtung“

Das Paket umfaßt die manuelle und automatische Ausrichtung. Die manuelle Ausrichtung umfaßt z.B. die Ausrichtung von Symbolen auf Kreisen oder in Ebenen. Hierfür ist die Klasse *GeometrieAusrichter* verantwortlich. Die automatische Ausrichtung erfolgt über simulierte Federsysteme. Für die Simulation ist die Klasse *Federsystem* zuständig. Auf Federsysteme soll im folgenden etwas näher eingegangen werden.

Ein Federsystem für die inneren Symbole eines äußeren Symbols umfaßt die folgenden Arten von Federn (vgl. auch Abbildung 12.14 auf nachfolgender Seite), deren Dimensionierung sich nach den Benutzereingaben im Fenster zur automatischen Ausrichtung richtet (vgl. Abschnitt 10.5.5).

1) Zwei Federn für jede Kante zwischen zwei inneren Symbolen dienen der Durchsetzung der gewünschten Kantenlänge (Art 1a) bzw. des gewollten Ebenenabstandes (1b). Im letzten Fall geht nur die Differenz der Y-Koordinaten der verbundenen Symbole in die Berechnung ein.

2) Eine Feder von jedem inneren Symbol zum Mittelpunkt des äußeren Symbols dient dazu, das innere Symbol nahe am Mittelpunkt zu halten, so daß kompakte Darstellungen erzeugt

werden können.

3) Eine Feder zwischen jedem Paar innerer Symbole sorgt dafür, daß Symbole voneinander weg streben. Abhängig von der aktuellen Entfernung zweier Symbole wird die entsprechende Feder unterschiedlich dimensioniert und zwar entweder mit der Konstante für den Minimalabstand zwischen Symbolen oder mit der Verteilungskonstante.

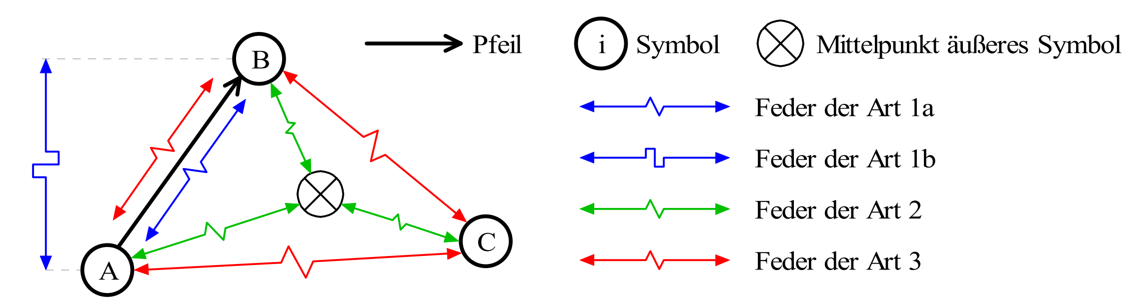


Abbildung 12.14: Aufbau eines Federsystems

Für die Simulation von Federsystemen stellen Ryall, Marks und Shieber einen Algorithmus vor [RMS97]. Diesem dient eine Menge von Knoten N und eine Menge von Federn S zwischen den Knoten als Eingabe. Für jeden Knoten n_i berechnet der Algorithmus mit $\underline{x}_i(t)$ die zeitabhängige Position[2]. Dies erfolgt anhand des ebenfalls zeitabhängigen Impulses $\underline{p}_i(t)$ an einem Knoten. Als Besonderheit wird – um ein unendliches Oszillieren des simulierten Federsystems zu verhindern – auch eine Dämpfung des Systems berücksichtigt, die über eine wählbare Dämpfungskonstante γ konfiguriert wird. Zur Vereinfachung wird die dabei eigentlich zu berücksichtigende Masse der Knoten einheitlich auf eins gesetzt.

Nachfolgend wird eine modifizierte Version des Algorithmus von Ryall, Marks und Shieber gezeigt, wie sie im J3Browser implementiert wurde:

$t = 0$;
fortfuehren = true;
forall $n_i \in N$: $\underline{p}_i(t) = \underline{0}$;
repeat {
    $t = t + \Delta t$;
    forall $n_i \in N$: $\underline{F}_i = \sum_{s \in S(i)} \underline{federkraft}(n_i, s, t)$;
    forall $n_i \in N$: {
        $\underline{p}_i(t) = \underline{p}_i(t - \Delta t) + (\underline{F}_i - \gamma\, \underline{p}_i(t - \Delta t))\Delta t$;

        if Position von n_i veränderbar:
            $\underline{x}_i(t) = \underline{kontrolle}(\ \underline{x}_i(t - \Delta t),\ \underline{p}_i(t)\Delta t\ )$;
        else
            $\underline{x}_i(t) = \underline{x}_i(t - \Delta t)$;
    }
    if letzte Bildschirmsaktualisierung länger als ΔT her {
        bildschirmAktualisieren();
        fortfuehren = *layoutListenerUpdate*();
    }
} until !*fortfuehren*;

In einem vorbereitenden Schritt werden die anfänglichen Impulse $\underline{p}_i(t{=}0)$ der Knoten initialisiert. Daran schließt sich die eigentliche Berechnung an. Hier wird zunächst jeweils die Simu-

[2] Unterstreichungen kennzeichnen hier dreidimensionale Vektoren bzw. vektorwertige Funktionen.

lationszeit um einen konstanten Wert Δt erhöht. Dann wird für jeden Knoten n_i die Gesamtkraft aller mit ihm verbundenen Federn ermittelt. Diese Federn werden durch $S(i)$ bezeichnet. Die Berechnung geschieht in Abhängigkeit von der Simulationszeit durch die vektorwertige Funktion $\underline{federkraft}$, auf die weiter unten noch eingegangen wird. Anhand der ermittelten Kräfte wird der auf jeden Knoten wirkende Impuls ermittelt und dann – wenn die Position eines Knotens veränderbar ist, d.h. das entsprechende Symbol z.B. nicht verankert ist – die Position aktualisiert.

Die Positionsaktualisierung ist beim J3Browser durch $\underline{kontrolle}$ einer Beaufsichtigung unterzogen. Im Regelfall gilt $\underline{kontrolle}(\underline{x},\underline{d}) = \underline{x}+\underline{d}$. Wenn aber die so ermittelten Koordinaten Maximalwerte über- bzw. Minimalwerte unterschreiten, so daß es z.B. zu einem Verlassen des übergeordneten Symbol kommen würde, kann eine Begrenzung auf eben diese Maximal- oder Minimalwerte erfolgen. Zusätzlich ist es möglich, eine Veränderung der z-Koordinate zu unterbinden, wenn nur ein zweidimensionales Layout gewünscht wird.

Gegenüber der Originalversion ist die Bedingung für die Terminierung des Algorithmus verändert. Bei Ryall, Marks und Shieber wird terminiert, wenn die kinetische Energie des Systems einen Schwellenwert unterschreitet. Beim J3Browser wird auf die Berechnung dieser Energie verzichtet. Vielmehr existiert ein sogenannter Layoutlistenern. Bei jeder Aktualisierung der Bildschirmanzeige, die alle ΔT Zeiteinheiten erfolgt, wird der Listener gefragt, ob das Layout fortgeführt werden soll. Derzeit sind zwei Listenertypen vorgesehen. Listener, die während der automatischen Ausrichtung verwendetet werden, fordern eine Terminierung, wenn der Benutzer dies über die Benutzungsschnittstelle verlangt, wohingegen Listener für die temporäre Einblendung eine vorgegebene Anzahl von Iterationen durchführen.

Es bleibt noch zu klären, wie die Kraft einer Feder s_j auf einen Knoten n_i mittels $\underline{federkraft}(n_i,s_j,t)$ berechnet wird. Aus der durch k_j bezeichneten Konstante der Feder, ihrer durch r_j bezeichneten Ruhelänge und ihrer sich aus der vorherigen Knotenpositionierung ergebenen Ausdehnung $l_j(t - \Delta t)$ ergibt sich der Betrag der Federkraft nach dem physikalischem Federgesetz mit $federkraft(n_i,s_j,t) = k_j(l_j(t - \Delta t) - r_j)$. Die Richtung der Kraft ist abhängig von der Art der Feder. Bei einer Federn zur Durchsetzung der Kantenlänge wirkt die Kraft bspw. entlang eines Vektors vom Knoten n_i zum Knoten, der durch die Feder mit n_i verbunden ist. Allgemein sei die Richtung durch $\underline{federrichtung}(n_i,s_j,t)$ bezeichnet, so daß sich ergibt:

$$\underline{federkraft}(n_i,s_j,t) = \underline{federrichtung}(n_i,s_j,t) * federkraft(n_i,s_j,t)$$

In der implementierten Version bekommt der Algorithmus die Mengen S und N nicht als Eingabe. Statt dessen bildet er diese implizit aus den im Diagramm enthaltenen Symbolen und Kanten. Zur Steigerung der Performanz konnte dabei nicht – wie zunächst vorgesehen – für jede Feder eine Instanz einer entsprechenden Java-Klasse *Feder* verwendet werden, sondern die Berechnung mußte in der Methode *layout* der Klasse *Federsystem* zusammengefaßt werden, die daher recht komplex wurde.

Damit die Aktualisierung der Position eines Symbols zunächst unabhängig von der Bildschirmanzeige erfolgen kann, ist jedem Symbol eine *LayoutInfo*-Instanz assoziiert, in der die Position zwischen gespeichert wird. Weiterhin umfaßt diese Instanz die auf das Symbol wirkende Kraft und den Impuls, sowie zwei Kennzeichen, die angeben, ob die Position des Symbols veränderbar und ob das Symbol sichtbar ist. Um die Performanz weiter zu steigern, sind all diese Informationen direkt, d.h. ohne Methodenaufruf, les- und schreibbar. Gleiches gilt für die layoutspezifischen Entitätselemente von Kanten. Jede Kante beinhalten diesbezüglich insbesondere Variablen für die Federkonstanten von Federn zur Durchsetzung der Kantenlänge und des Ebenenabstands. Diese Federkonstanten sind eine Addition der sich aus den verschiedenen Pfeilen der Kante ergebenen Konstanten.

12.8 Paket „darstellung.graphik“

Aus der Sicht des J3Browsers realisiert dieses Paket die dritte Stufe des Dreistufenmodells, da es den VRML-Browser und das EAI kapselt. Damit steuert dieses Paket zwar kein physisches Ausgabegerät an, hardwarenähere Aspekte liegen aber nicht mehr in der Verantwortung des J3Browsers.

Ein wichtiges Ziel beim Entwurf des Systems war es, die Realisierung der graphischen Ausgabe möglichst vollständig zu kapseln, da während der Experimentierphase erkannt werden mußte, daß die verwendete Kombination aus VRML-Szenen und einem Java-Applet zwar für prototypische Entwicklungen taugt, für eine Produktreife aber ein Umstieg angebracht wäre. Bei der Verwendung von VRML und dem EAI mußten folgende Probleme berücksichtigt werden, auf die hier näher eingegangen werden soll:

- Performante graphische Realisierung von Diagrammen
- Stabile Ereignisbehandlung
- Performate Aktualisierung der Darstellung
- Speichereffiziente Szenengraphen

12.8.1 Performante graphische Realisierung von Diagrammen

Beim initialem Erzeugen und bei jedem Einladen eines Diagrammes muß eine VRML-Szene erstellt werden, die zur Anzeige des Diagrammes dient. Dies muß genügend performant geschehen, um den Betrachter nicht durch lange Wartezeiten abzuschrecken.

Das EAI stellt eine Methode *createVRMLFromString* zur Verfügung, die als Parameter VRML-Quelltext erwartet. Aus diesem Quelltext wird ein Szenengraph erzeugt, der über weitere Methoden bspw. in den aktuell dargestellten Szenengraph eingefügt werden kann. Dieser Mechanismus wird für den J3Browser genutzt, wobei der dargestellte Szenengraph initial weitgehend leer ist. Zu klären war, wie der Quelltext aus der internen Repräsentation des Diagrammes abzuleiten ist. Während der Experimentierphase wurde jeder benutzte VRML-Knotentyp durch eine Klasse gekapselt. Es lag in der Verantwortung der Diagrammteile, diese nach Bedarf zu instanziieren und zu attributieren. So wurde ein Abbild des späteren Szenengraph als Java-Objektstruktur geschaffen. Die Klassen für Knotentypen verfügten über Methoden zur Quelltexterzeugung. Über einen Durchlauf durch die Objektstruktur wurde bei jedem Laden mit Hilfe dieser Methoden der Quelltext für das Diagramm gewonnen.

Obige Lösung hat sich aber als nicht hinreichend performant erwiesen. Beispielsweise dauerte das Laden eines Diagrammes der Java-AWT-API, dessen VRML-Quelltext mehrere Megabytes umfaßte, auf dem Entwicklungsrechner über fünf Minuten. Auch wenn der VRML-Quelltext für ein Diagramm nur einmal erzeugt und dann immer wieder beim Einladen des Diagramms verwendet wird, sinkt die Dauer nicht sonderlich stark. Der Grund dafür ist, daß ein Großteil der Zeit dafür aufgebracht werden muß, den sehr großen Szenengraph für das Applet verfügbar zu machen. Weniger zeitintensiv ist das Laden, wenn Prototypen für Diagrammteile verwendet werden und so der bereits für ein einzelnes Teil recht umfangreiche Teilgraph stark verkleinert werden kann. Es wurden deshalb Prototypen für die verschiedenen Diagrammteile sowie für größere Ausschnitte aus den Teilen – wie z.B. Eigenschaftsmarkierungen – definiert. Dies erfolgt in einer VRML-Datei „*Protos.wrl*“, die dementsprechend zum *graphik*-Paket zu zählen ist. Das gilt auch für eine weitere VRML-Datei mit Namen „*Leerszene.wrl*“, welche die inital geladene – und weitgehend leere – Szene enthält, die später dann mit dem Diagramm gefüllt wird. Diese Datei ist, wie das Darstellungsapplet selbst, in die HTML-Datei „*Darstellung.html*“ eingebettet.

Man könnte es bei der vorgestellten Lösung problematisch finden, daß die Definition des Aussehens eines Teils nicht mehr Bestandteil der entsprechenden *Teil*-Klasse ist. Demgegenüber ergibt sich zusätzlich zum Performanzvorteil eine stärkere Trennung des *diagramm*-Paketes vom EAI. Während es beim zuerst beschriebenen Verfahren für *Teil*-Klassen notwendig war, mit Abbildungen von VRML-Knotentypen zu operieren, wird von *graphik*-Paket jetzt eine abstraktere Sicht angeboten: es stellt Klassen zur Verfügung, welche die Ausgabe von Diagrammteilen zusammen mit dem Erkennen von Manipulationen des Benutzers an diesen kapseln. Gemeinsame Superklasse dieser Entitäten ist *GraObj*. Bei einem Wechsel der Anzeigekomponente brauchen nur hier Veränderungen vorgenommen werden, wohingegen *Teil*-Klassen unverändert bleiben können.

Zusammen mit der weiter unten beschriebenen Maßnahme zur Beschleunigung der Szenenmanipulation wurde so erreicht, daß z.B. die AWT-Visualisierung nun weitaus schneller eingeladen werden kann. Auch nach dem Erzeugen der VRML-Szene werden aber gelegentlich noch weitere graphische Objekte benötigt, bspw. dann, wenn neue Gruppen definiert werden. Um hier zu beschleunigen, werden initial mehr Objekte erzeugt als gebraucht werden. Ansonsten müßte der VRML-Browser zu häufig die Datei „*Protos.wrl*" verarbeitet werden. Alle Objekte werden durch die Singleton-Klasse *GraObjPool* zwischengespeichert. Bei Bedarf werden sie diesem Objektpool dann entnommen. Es kann auch dazu kommen, daß graphische Objekte nicht mehr weiter benötigt werden. Dann werden sie erneut in den Pool eingefügt und können so wiederverwendet werden.

Die graphische Realisierung eines Diagramms läuft unter diesen Voraussetzungen wie folgt ab: Nachdem die interne Repräsentation des Diagramms aufgebaut wurde (vgl. Abschnitte 12.5.2 und 12.5.3), erfolgt zunächst eine Reservierung der benötigten graphischen Objekte beim Objektpool. Dazu wird für jedes Diagrammteil die Methode *reserviereGraObj* aufgerufen. Die verschiedenen Subklassen von *Teil* redefinieren diese Methode so, daß der Objektpool darüber informiert wird, welcher Art die jeweils vom Teil benötigten graphischen Objekte sind. Der Objektpool sammelt diese Informationen und weiß daher am Ende, wie oft jede Art gebraucht wird. Er erzeugt dann einige Objekte mehr als benötigt. In einem zweiten Durchlauf durch das Diagramm wird für alle Diagrammteile *realisiere* aufgerufen. Jedes Diagrammteil besorgt sich jetzt die notwendigen graphischen Objekte vom Objektpool und konfiguriert diese entsprechend der darzustellenden Gegebenheit. Das heißt bspw., daß Symbole die Beschriftungen entsprechend dem Namen des symbolisierten Elements setzen oder daß, je nachdem welcher Sonderfall bei einer Benutzungsbeziehung vorliegt, eine passende Pfeilspitze ausgewählt wird.

12.8.2 Stabile Ereignisbehandlung

Über das EAI ist es möglich, daß ein Applet über Aktionen des Benutzers in der Szene informiert wird. Insbesondere die Benachrichtigung über Ereignisse, die durch *TouchSensor*-Knoten ermittelt werden, d.h. also Mausereignisse, hat sich als instabil erwiesen. Es kam häufig zu Systemabstürzen, wenn während der Reaktion auf ein Ereignis bereits ein neues Ereignis auftrat. Für diesen beim J3Browser häufigen Fall mußte eine Maßnahme zur Stabilisierung gefunden werden.

Es wurde mit der Singleton-Klasse *EventQueue* eine Entkopplung der Benachrichtigung durch das EAI und der Reaktion auf die Nachricht vorgenommen. Bei einer Nachricht wird lediglich ein Eintrag in einer durch *EventQueue* verwalteten Warteschlange vorgenommen. Dies kann in so kurzer Zeit geschehen, daß weitere Ereignisse in der Zwischenzeit nicht zu erwarten sind. Ein ebenfalls in *EventQueue* gekapselter Thread prüft, ob Einträge in der Warteschlange vorhanden sind und bearbeitet diese ggf, wobei dann – u.U. zeitintensive – Reaktionen des J3Browsers ausgeführt werden. Die Stabilität konnte so um einiges gesteigert

werden.

12.8.3 Performante Aktualisierung der Darstellung

Eine Aktualisierung der Darstellung kann aus zwei Gründen notwendig werden: zum einen dadurch, daß der Betrachtungsstandort oder der Betrachtungswinkel verändert wird und zum anderen durch Manipulationen an der Szene selbst.

Bei Änderungen am Betrachtungsstandort oder -winkel muß die Bildschirmdarstellung so schnell aktualisiert werden, daß für den Betrachter der Eindruck einer flüssigen Bewegung entsteht. Dies ist weitgehend die Aufgabe des VRML-Browsers, der dazu auf ggf. vorhandene spezielle Hardware zurückgreift. Bei sehr komplexen Szenen kann es aber notwendig sein, hier unterstützend zu wirken. Bei J3Browser werden daher an vielen Stellen *LOD*-Knoten eingesetzt, um ein Zeichnen weit entfernte Objekte, die vom Betrachter nicht mehr erkannt werden können, im Sinne einer Elision zu verhindern. Beispielsweise werden so Schatten, Beschriftungen und Eigenschaftsmarkierungen von weit entfernten Symbolen abgeschaltet.

Bei Manipulationen der Szene müssen Attribute der Szenengraphknoten geändert werden. Das EAI besitzt die Eigentümlichkeit, daß derartige Veränderungen abhängig vom Kontext mit stark unterschiedlicher Performanz durchgeführt werden. Erfolgt die Veränderung bspw. Aufgrund einer Aktion in der Benutzungsschnittstelle, wird durch den VRML-Browser mit jedem Aufruf einer EAI-Methode die Darstellung aktualisiert. Da meist sehr viele Aufrufe notwendig sind, dauert die gesamte Manipulation inakzeptabel lange. Erfolgt die Veränderung jedoch als Reaktion auf ein durch das EAI gemeldetes Ereignis, wird die Darstellung erst dann aktualisiert, wenn die entsprechende Reaktion vollständig abgearbeitet wurde. Dies führt dazu, daß die Dauer derselben Veränderung kontextabhängig im Bereich von einigen Minuten liegt kann oder auch so kurz sein kann, daß sie kaum wahrnehmbar ist.

Um Veränderungen zu beschleunigen wurde deshalb die Singleton-Klasse *ActionManager* zusammen mit deren innerer Schnittstelle *Action* geschaffen. Für Veränderungen wird ein Objekt einer Klasse erzeugt, die *Action* implementiert. Dieses Objekt wird dem *Action-Manager* übergeben, der daraufhin ein EAI-Ereignis initiiert und als Reaktion darauf die im übergebenen Objekt gekapselten Veränderung durchführt. Aufgrund der beschriebenen Instabilität muß dieser Ablauf variiert werden, wenn bereits eine Ereignisbehandlung aktiv ist. Dann wird die Veränderung ohne neues EAI-Ereignis direkt durchgeführt.

12.8.4 Speichereffiziente Szenengraphen

Ein weiteres Problem wurde bei der vorliegenden prototypischen Implementierung nicht im vollen Umfang gelöst: bei der Verarbeitung des Szenengraphen durch den VRML-Browser besteht ein immenser Speicherbedarf. Bei der angesprochenen AWT-Visualisierung werden über 100 Megabytes benötigt. Hier wird somit der derzeit mögliche Visualisierungsumfang begrenzt.

Eine Abhilfe könnte in der häufigeren Mehrfachverwendung von Knoten über den DEF/USE-Mechanismus liegen. Da Diagrammteile aber unabhängig voneinander editiert werden können, wäre eine relativ komplexe Verwaltung von Knoten notwendig. Um Speicherplatz zu sparen wäre es z.B. sinnvoll, daß alle gleich aussehenden Diagrammteile gemeinsam einen Knoten vom VRML-Typ *Shape* verwenden, der das Aussehen bestimmt. Dieser Knoten wäre mehreren *Transform*-Knoten untergeordnet, um den Knoten für verschiedene Diagrammteile an verschiedenen Positionen zeigen zu können. Es ist jetzt aber zu beachten, daß, wenn eines der Diagrammteile z.B. die Farbe wechseln soll, der Szenengraph umorganisiert werden müßte. Es müßte dann ein neuer *Shape*-Knoten erzeugt und dem *Transform*-Knoten des veränderten Diagrammteils untergeordnet werden, nachdem der dort zuvor vorhandene gemeinsame *Shape*-Knoten entfernt wurde.

Erschwerend kommt hinzu, daß innerhalb der Definition eines Prototyps nicht auf Knoten außerhalb des Prototyps verwiesen werden kann. Der Verweis kann allein durch die Belegung der im Kopf des Prototyps deklarierten Attribute erfolgen.

Insgesamt wurde daher von einer ausgiebigen gemeinsamen Verwendung von Knoten abgesehen, wobei aber der DEF/USE-Mechanismus innerhalb von Prototypen nach Möglichkeit genutzt wird.

12.9 Übersicht über die Realisierung der Visualisierungstechniken

Als Ausgangspunkt für spätere Weiterentwicklungen des J3Browsers wird an dieser Stelle noch einmal zusammengetragen, wo sich Realisierungen der im Kapitel 8 vorgestellten Visualisierungstechniken finden lassen.

Da sind zunächst die Klassen *ConeTree*, *Kegel* und *Baum* zu nennen, mit deren Hilfe die verschiedenen Formen von Anordnungen umgesetzt werden. Für die sichtbare Drehung von Cone Trees und Kegeln ist die Klasse *Drehmaschine* verantwortlich. Die Gruppierung der zu einer Anordnung gehörenden Symbole ist Aufgabe der Klasse *AnordnungsgruppenSymbol*, wohingegen allgemeine Gruppen durch *GruppenSymbol* implementiert werden.

Die für die Darstellung von Paketzugehörigkeiten, Pakethierarchien und Entitätsschachtelungen einsetzbaren Information Cubes werden innerhalb der Klassen *Teil* und *Symbol* umgesetzt. Symbole sind deswegen darauf vorbereitet, innere Diagrammteile zugeordnet zu bekommen.

Um der u.U. schlechten Erkennbarkeit von Beziehungen zwischen Entitäten verschiedener Pakete entgegenzutreten, wurden Paketzusammenfassungen, die temporäre Symboleinblendung sowie die Anzeige von Pfeilbeschreibungen umgesetzt. Für Paketzusammenfassungen ist die Klasse *ZusammenfassungsSymbol* verantwortlich. Die Symboleinblendung erfolgt durch die Klasse *EinblendungsManager*. Die Anzeige von Pfeilbeschreibungen geschieht in der Statuszeile, welche dazu vom Berührungsmanager über Pfeilberührungen unterrichtet wird (Klassen *Statuszeile* und *BeruehrungsManager*).

Die Pfeile zur Darstellung von Abhängigkeiten zwischen verschiedenen Elementen werden über die Klasse *PfeilErzeuger* erzeugt. Für die Degree-of-Interest-Darstellung von Pfeilen ist die Klasse *DOIManager* verantwortlich. Ein letzter Manager, der *DokuManager*, sorgt für die Darstellung von Quelltextkommentaren in der dreidimensionalen Szene.

Für die Ausrichtung von Symbolen ist das Paket *darstellung.diagramm.ausrichtung* verantwortlich, wobei die Klasse *GeometrieAusrichter* für geometrische Ausrichtungen auf Kreisen oder ähnliches zuständig ist, während die Klasse *Federsystem* die automatische Ausrichtung mittels Federsystemsimulationen realisiert. Hierdurch kann auch eine ebenenförmige oder top-down-Darstellung erreicht werden. Gleichzeitig ist die Klasse *Federsystem* für die Aufwandsbegrenzung beim Anfertigen von Visualisierungen zentral, da es durch sie möglich wird, unaufwendig halbwegs geeignete Ansichten zu erstellen.

Das Filtern von Symbolen und Pfeilen eines Diagramms wird über das Konzept der effektiven Sichtbarkeit von Diagrammteilen sowie über die Klasse *FilterSpezifikation* realisiert.

Die Notation schließlich wird im Zusammenspiel mehrerer Klassen umgesetzt. Dies sind zum einen die Klasse *Teil* und ihre Subklassen sowie ebenso die Klasse *GraObj* und deren Subklassen. Letztere stützen sich auf die in der Datei „*protos.wrl*“ definierten Prototypen, um das Aussehen der Diagrammteile über VRML sichtbar zu machen. Erste sind weitgehend unabhängig von der verwendeten Anzeigetechnologie und bilden die Eigenschaften der Gegebenheiten eines Strukturmodells in die Notation ab.

Teil IV

Abschluß

13 Bewertung und Ausblick

Ziel dieser Arbeit war es, eine Konzept für eine dreidimensionale Visualisierung der Strukturen innerhalb von Java-Software aufzuzeigen und dieses Konzept prototypisch umzusetzen.

Das entwickelte Konzept beruht insbesondere auf einer flexiblen Kombination verschiedener zwei- und dreidimensionaler Visualisierungstechniken, wie z.B. der Cone Trees oder Information Cubes. Es zeigte sich, daß Sprachkonzepte von Java, wie z.B. Klassen oder Vererbung, nicht starr auf bestimmte Darstellungstechniken abgebildet werden sollten, sondern eine Lösung vorzuziehen ist, die es dem Anwender erlaubt, Techniken gemäß der Eigenschaften des zu zeigenden Sachverhaltes auszuwählen.

Als Basis der Konzeptumsetzung durch den entwickelten J3Browser wurde VRML zusammen mit den External Authoring Interface (EAI) für Java-Applets verwendet. Die Vorteile dieser Lösung wurden schon verschiedentlich genannt. So konnte insbesondere auf die Implementierung eigener Navigationsfunktionalität verzichtet werden und es war wegen der textuellen Beschreibung von Szenen leicht, Ideen auszuprobieren. Die Verwendung von Java erlaubte es, eine graphische Benutzungsschnittstelle in den J3Browser zu integrieren, was nur mit VRML und ohne das EAI nicht möglich gewesen wäre.

Im Verlauf der Arbeit erwies sich die Softwarebasis aber als problematisch. Die Umsetzung wurde durch eine große Unzuverlässigkeit des EAI mit geprägt. Viel Arbeitsaufwand mußte betrieben werden, um dieser zu begegnen. Eine Reihe von Maßnahmen wurden bei der Besprechung des Entwurfs vorgestellt. So wurde mit der Verarbeitung der über das EAI gemeldeten Ereignisse in einem eigenständigen Thread versucht, der großen Instabilität des EAI bei der Ereignisverwaltung zu begegnen. Über die Verwendung von VRML-Prototypen mußte die ansonsten nicht ausreichende Performanz der graphischen Realisierung von Diagrammen verbessert werden. Auch die Manipulation der entstehenden dreidimensionalen Szenen galt es zu beschleunigen. Dadurch, daß nun immer alle zu einer Veränderung gehörenden Aktionen in einem Objekt zusammengefaßt werden, wurde es möglich, diese Aktionen in einem Zug und ohne zwischenzeitliche Bildschirmaktualisierung durchzuführen, was zu einer starken Beschleunigung führte.

Einige Probleme blieben aber bestehen, so z.B. der hohe Speicherbedarf für Szenengraphen und auch eine Unzulänglichkeit des HTML-Browsers bei Änderungen der Bildschirmfenstergröße, die dazu führt, daß die Darstellung nicht immer vollständig aktualisiert wird. Schwerwiegender ist es, daß auch durch den eigenständigen Thread die Ereignisverwaltung nicht vollständig stabilisiert werden konnte. Wenn gleichzeitig mit der Durchführung längerer Manipulationen Ereignisse auftreten, kann es ab und an immer noch zu Abstürzen kommen. Die Frage, inwieweit diese Probleme durch die Verwendung einer anderen Implementierung des EAI, d.h. also eines anderen VRML- und/oder eines anderen HTML-Browser, hätten beseitigt werden können, mußte offen bleiben, da es trotz der für das EAI postulierten Plattformunabhängigkeit nicht gelang, den J3Browser ohne weiteres mit anderen Browser-Kombinationen als dem Netscape Communicator 4.6 und den Cosmo-VRML-Browser 2.1 laufen zu lassen.

Ein Umstieg auf eine andere Technologie zur Anzeige von dreidimensionalen Szenen, wie z.B. auf die Java3D API, auf OpenGL oder Direct3D, dürfte somit langfristig – vielleicht im Hinblick auf ein System mit Produktreife – erforderlich sein. Durch das weitgehende Kapseln der Anzeigetechnologie erscheint dies allerdings relativ leicht möglich.

Obwohl die verwendete Softwareplattform mit einigen Fallstricken versehen ist, eignet sich

der entwickelte J3Browser doch dafür, einen guten Eindruck von der Leistungsfähigkeit dreidimensionaler Visualisierungen zu bekommen. Um einen direkten Vergleich zwischen zwei- und dreidimensionalen Diagrammen zu ermöglichen, wurden von einigen der während der Entwicklung erstellten Darstellungen auch zweidimensionale Versionen erstellt (vgl. Anhang A). Auch hierfür konnte der J3Browser eingesetzt werden. Beschränkt man sich bei der Navigation durch eine solche Szene bei direktem Blick zur Ebene auf ein Gleiten parallel der Ebene sowie auf ein gerades auf die Ebene zu- bzw. von dieser wegtreten, kann der Bildlauf und das Zooming zweidimensionaler Systeme simuliert werden. Allerdings besteht dann immer noch eine Nahtlosigkeit der Bewegung, wie sie von gängigen Systemen – wie bspw. Rational Rose [RATIONAL] – häufig nicht geboten wird.

Einen derartigen Vergleich in der Form einer empirischen Studie mit einer Gruppe von Probanden – bspw. Softwareentwicklern aus der Praxis – durchzuführen, würde eine sinnvolle Ergänzung zur vorliegenden Arbeit darstellen. Als Ansatz dazu könnte das *Path Tracking Problem* dienen (vgl. Kapitel 2). Die detaillierte Konzeption und Verwirklichung einer solchen Studie hätte aber den Rahmen dieser Arbeit gesprengt, so daß sich nachfolgende Betrachtungen auf eine Wiedergabe subjektiver Erfahrungen beschränken muß.

Eine dieser Erfahrungen ist es, daß ein Einhalten obiger Navigationsrestriktionen schwer durchzuhalten ist. Häufig kommt es zu einem Wechsel des Blickwinkels, vergleichbar einem Blick nach rechts oder links. Dies kann als ein Hinweis auf die Nützlichkeit der zusätzlichen Navigationsmöglichkeiten im dreidimensionalen Raum und insbesondere der perspektivischen Verkleinerung gewertet werden, denn durch den Seitenblick wird diese auf die Ebene mit dem Diagramm angewendet und mehr vom Gesamtzusammenhang kommt ins Blickfeld.

Eine weitere Erfahrung ist es, daß es im Dreidimensionalen tatsächlich schnell zu einem Gefühl des in-die-Szene-hinein-versetzt-sein kommt – ein Eindruck, der bei der flächigen Darstellung vollständig entfällt. Während der Wanderungen durch die visualisierte Software verdichten sich die dargestellten Vererbungshierarchien, Paketeinteilungen usw. zu einem Gesamtbild. Dabei hilft das räumliche Erinnerungsvermögen und die perspektivische Verzerrung, die Orientierung zu bewahren. Man wird zu ausgiebigen Rundgängen motiviert, die zu einer intensiven Beschäftigung mit der Software führen. So wurden dadurch z.B. während dieser Arbeit eine Reihe von Zusammenhängen in der komplexen Java-Klassenbibliothek aufgedeckt, die dem Autor – trotz einiger Jahre praktischer Arbeit mit der Bibliothek – noch unbekannt waren. Dabei hat sich vor allem auch die Integration der Dokumentation in die dreidimensionale Darstellung bewährt.

Es zeigt sich jedoch auch, daß der Übergang von zwei- zu dreidimensionalen Visualisierungen allein nicht zwangsläufig immer zu expressiven Darstellungen führt. Die gleichzeitige Darstellung aller Gegebenheiten einer Software ist auch hier oft nicht möglich. Die Größe darstellbarer Sachverhalte bleibt begrenzt, wenn die Grenze auch erweitert wird, wie dies z.B. die Verwendung von Cone Trees für Klassenhierarchien zeigt. Durch die schiere Vielzahl der Elemente und Beziehungen in nicht trivialer Software bleiben weitere Mechanismen notwendig, die auch im Zweidimensionalen möglich und nützlich sind, wie z.B. die Gruppenbildung oder die Verwendung von Filtermechanismen.

Zwei Nachteile dreidimensionaler Visualisierungen müssen genannt werden. Beide sind allerdings nicht prinzipieller Natur, sondern beziehen sich auf mangelnde Hardwareunterstützung. Zum einen ist das Fehlen eines Tiefeneindrucks bei der Projektion der Szene auf dem Bildschirm ein Problem, das alleine durch Schatten nicht ausgeräumt werden kann. Zwar kann mit einer permanenten Bewegung der Szene gut ein Gefühl von Tiefe vermittelt werden, die damit einhergehende ständige Veränderung der Anzeige erschwert es aber, sich auf ausgewählte Bereiche der Darstellung zu konzentrieren, da es oft

schon nicht einfach ist, diese im Blick zu behalten. Es ist aber zu erwarten, daß Eye-Shutter-Brillen oder dergleichen dieses Problem beheben. Leider stand während dieser Arbeit derartiges nicht zur Verfügung. Ein weiteres Problem ist es, daß die Navigation im dreidimensionalen Raum, für die ja eigentlich eine Natürlichkeit erwartet wurde, mit herkömmlichen – zweidimensionalen – Eingabegeräten, wie Maus oder Joystick, anfänglich schwierig ist und so erst nach einer gewissen Einarbeitungszeit halbwegs vernünftig navigiert werden kann. Es gibt aber auch hier Entwicklungen, von denen zu erwarten ist, daß sie Abhilfe schaffen. Geräte, wie z.B. Datenhandschuhe etc., sind derzeit aber noch vergleichsweise kostenintensiv.

Konkreter auf den J3Browser bezogen ist die Erkenntnis, daß das implementierte Federmodell vielfach Ergebnisse liefert, deren weitere Bearbeitung sich auf wenige Korrekturen beschränken oder auch ganz entfallen kann, zumindest wenn man nur auf einen ersten Überblick über eine Software abzielt. Dabei ist es auffällig, daß bei einer Begrenzung der Federmodellanwendung auf zwei Dimensionen, wie sie mit dem J3Browser möglich ist, Darstellungen entstehen, die weniger ansprechend und übersichtlich erscheinen, z.B. weil sie weit mehr Überschneidungen von Pfeilen enthalten.

Das Federmodell bietet aufgrund seiner Flexibilität eine Vielzahl von Erweiterungsmöglichkeiten. Man kann sich die Modellierung unterschiedlichster Federn vorstellen, durch deren Anwendung auch Beziehungen zwischen Elementen hervorgehoben werden können, die in der derzeitigen Implementierung noch keine Berücksichtigung finden. Beispielsweise könnte man nach einer Erweiterung der Analyse und des Metamodells für Strukturmodelle Federn zwischen den Entitäten einer Quelltextdatei vorsehen und so diesen Zusammenhang betonen, da es, wie die Softwarebeispiele zeigen, zu erwarten ist, daß Entitäten zwischen denen ein besonders starker Bezug besteht, in einer Datei deklariert werden. Allerdings kann die gemeinsame Nutzung einer Datei nicht über Doclets festgestellt werden.

Als Fernziel ist eine Abfragesprache denkbar, mit der Elemente der visualisierten Software selektiert werden können, zwischen denen dann Federn gespannt werden. Dem Betrachter würde so die Möglichkeit gegeben, die Darstellung schnell auf sein gegenwärtiges Informationsbedürfnis anzupassen, ohne auf vorgegebene Auswertungsmöglichkeiten beschränkt zu sein. Beispielsweise könne man, wiederum bei erweiterter Analyse und erweitertem Metamodell, Abfragen ermöglichen, bei denen alle Entitäten selektiert werden, die eine speziellen Methode einer ausgewählten Klasse aufrufen. So können Erkenntnisse darüber gewonnen werden, wo ein Abändern von Methoden besonders zu berücksichtigen ist.

Trotz der guten Eignung des Federmodells ist der Einsatz von Bäumen, Cone Trees und Kegeln zur Akzentuierung von wichtigen Hierarchien aber weiterhin sinnvoll. Es entstehen so prägnante Bereiche in der Darstellung, die mit Landmarken im natürlichen Gelände vergleichbar sind und damit die Wiedererkennung und Orientierung unterstützen. Zudem ist gerade bei Cone Trees eine Interaktionsform gegeben, bei der das Fehlen dreidimensionaler Eingabegeräte nicht allzusehr ins Gewicht fällt.

Eine vergleichbare leichte Navigationsmöglichkeit bietet auch die Kugelprojektion bei der vom Betrachter eine Kugel zu rotieren ist (vgl. Seite 40). Die Kugelprojektion wurde zunächst als eine Alternative zur Information-Cubes-Technik in Betracht gezogen. Pakete wären als drehbare Kugeln dargestellt worden, auf deren Oberfläche die Symbole der zugehörigen Entitäten anzuordnen gewesen wären. Auch hier könnte eine Kombination mit dem Information-Landscape-Ansatz erfolgen, indem mehrere Kugeln in einer Landschaft positioniert werden. Ein Probleme dieses Ansatzes ist die mangelnde Kombinierbarkeit insbesondere mit Cone Trees oder Kegeln. Da die Verwendung der Information-Cubes-Technik deshalb bei weitem vorteilhafter erschien, wurde auf die Aufnahme der von der Implementierung relativ komplexen Kugelprojektion in den Klassenbrowser verzichtet.

Besonders die Realisierung gekrümmter Pfeile, wie sie im Conformal-Model vorgesehen sind, wäre mit VRML nur schwer zu machen gewesen. Zudem bietet VRML zwar eine Möglichkeit, das Drehen von graphischen Objekten über Ereignisse zu erkennen, es zeigt sich aber, daß bei der verwendeten Browser-Kombination Drehungen fehlerhaft gemeldet werden.

Die direkte Unterstützung von Cone Trees kann als Vorteil gegenüber NV3D gewertet werden. Eine vergleichbare Interaktionsform steht dort nicht zur Verfügung. Eine den Cone Trees entsprechende statische Anordnung von Symbolen kann zwar bei NV3D manuell erzeugt werden – wenn auch ohne halbtransparente Kegel – dies dürfte aber sehr mühsam sein. Auch ArchView bietet keine Cone Trees und zudem keine Information Cubes. Die Fähigkeit von ArchView mit Relationen zu rechnen dürfte aber eine sinnvolle Ergänzung des J3Browsers sein, gerade auch um die hohe Anzahl von dargestellten Benutzungsbeziehungen sinnvoll zu verringern.

Die Integration der verschiedenen Visualisierungstechniken mit dem Federmodell ist beim derzeitigen System noch ausbaufähig. So könnte man sich vorstellen, daß auch die Positionierung eines Cone Trees in der Darstellung auf diesem Wege automatisiert wird. Einige Ansätze hierfür finden sich in [RMS97]. Dort wird gezeigt, wie mit Hilfe von Federn einfache Anordnungen von Symbolen durchgesetzt werden, bspw. ein Ausrichten in einer Ebene oder das gleichmäßige Verteilen auf einer Strecke. Es wäre zu prüfen, ob sich diese Ansätze auf komplexe Anordnungen übertragen lassen.

Auch in anderer Hinsicht gibt es noch einige Erweiterungsmöglichkeiten beim J3Browser. Neben weiteren Verbesserungen der Editierfunktionen – z.B. wäre eine *Undo*-Funktion zweckdienlich – könnte an einigen Stellen durch Nutzung von Animation die Objektkonstanz verbessert werden. Insbesondere wäre dies bei der Ikonifizierung und Deikonifizierung von Gruppen sinnvoll. Weiterhin könnte die Möglichkeit geschaffen werden, virtuelle Rundflüge durch Diagramme in der Form geführter Touren zu entwerfen. An bestimmten Stellen würden dann z.B. vorgegebene Dokumente angezeigt. So würde die Eignung des J3Browser zur Dokumentation von Softwaresystemen weiter verbessert.

Unabhängig vom J3Browser kann nach Ansicht des Autors abschließend jedoch festgehalten werden, daß trotz der genannten Probleme ein großes Potential für die Anwendung dreidimensionaler Visualisierung von Softwarestrukturen besteht. Für die Zukunft ist infolgedessen mit einem umfassenden praktischen Einsatz zu rechnen, zumal von einer größeren Verbreitung unterstützender Hardware ausgegangen werden kann. Weitere Forschungs- und Entwicklungsbemühungen in diesem Bereich dürften daher sehr lohnend sein.

Der Autor wird sie mit großem Interesse verfolgen.

14 Literatur

[AF00] K. ALFERT, A. FRONK: *3-Dimensional Visualization of Class Relations*. Beitrag zu 2000 IDPT Conference - The Fifth World Conference on Integrated Design & Process Technology, Dallas, URL: http://ls10-www.informatik.uni-dortmund.de/~alfert/publications/idpt2000.pdf, 2000.

[AG98] K. ARNOLD, J. GOSLING: *The Java Programming Language*. Addison-Wesley, Amsterdam, 1998.

[Alf99] K. ALFERT: *Developing the Altenberger Dom Presentation - Integrating Content Providers and Software Developers.* In: W. Hahn, E. Walter-Klaus, J. Knoop (eds): Euromedia'99, S. 70-77, Munich, Germany, URL: http://ls10-www.informatik.uni-dortmund.de/~alfert/publications/euromedia99.pdf, 1999

[And95] K. ANDREWS: *Visualising cyberspace: Information visualization in the harmony internet browser*. In: Proc. of First IEEE Symposium on Information Visualization, Atlanta, GA, S. 97-104, Okt. 1995.

[ANM97] A. L. AMES, D. R. NADEAU, J. L. MORELAND: *VRML 2.0 Sourcebook*, 2nd Edition, John Wiley & Sons, Inc., New York u.a., 1997.

[APW96] K. ANDREWS, M. PICHLER, P. WOLF: *Towards rich information landscapes for visualising structured web space*. In: Proc. of 2nd IEEE Symposium on Information Visualization, Info Vis'96, San Francisco, CA, Oct. 1996.

[BET+94] G. D. BATTISTA, P. EADES, R. TAMASSIA, IOANNIS G. TOLLS: *Algorithms for drawing graphs: an annotated bibliography*. In: Computational Geometry: Theory and Applications, vol. 4 no. 5, S. 235-282, 1994.

[BF96] I. BRUß, A. FRICK: *Fast interactive 3-d graph visualization*. Graph Drawing (Proc. GD '95). LNCS, 1027:99-110, Springer-Verlag, 1996.

[Boo94] *G. BOOCH: Objektorientierte Analyse und Design,* Addison-Wesley, Bonn u.a., 1994.

[BRJ99] G. BOOCH, J. RUMBAUGH, I. JACOBSON: *Das UML-Benutzerhandbuch*. Addison-Wesley, Bonn, 1999.

[Bro88] M. H. BROWN: *Exploring Algorithms Using Balsa II*. In: IEEE Computer, 21(5): 14-36, 1988.

[Bro92] K.W. BRODILIE [HRSG.]: *Scientific Visualization - techniques and applications*. Springer-Verlag, Berlin u.a., 1992.

[BS84] M. H. BROWN, R. SEDGEWICK: *A System for Algorithm Animation.* In: Proceedings of ACM SIGGRAPH `84, (S. 177-186). ACM, New York, 1984.

[C4W] Homepage der Firma *C4W.COM, Inc*. URL: http://www.c4w.com

[COSMO] *Cosmo Player 2.1*. Copyright 2000 by Computer Associates International, Inc. Islandia, USA, URL: www.cai.com/cosmo.

[CT96] I. F. CRUZ AND J. P. TWAROG. *3d graph drawing with simulated annealing*. Graph Drawing (Proc. GD '95). LNCS, 1027:162--165, Springer-Verlag. 1996.

[DH96] R. DAVIDSON, D. HAREL: *Drawing Graphs Nicely Using Simulated Annealing.* In: ACM Transactions on Graphics, Vol. 15, No. 4, October 1996, Pages 301–331.

[Eng93] HERMANN ENGESSER (LTG. HRSG): *Duden "Informatik": ein Sachlexikon für Studium und Praxis*. 2., vollst. überarb. und erw. Auflage, Mannheim u.a., Dudenverlag, 1993.

[Eng95] H. ENGLBERGER: *Computergestützte Informationsvisualisierung – Eine Klassifikation aktueller Techniken und ihre Einsatzpotentiale für die Unternehmung*. Diplomarbeit, TU München, Fakultät Informatik, Nov. 1995.

[ES90] *M. A. ELLIS, B. STROUSTRUP: The Annotated C++ Reference Manual*, Addison-Wesley, , Reading, Massachusetts, U.S.A., 1990.

[FDF+90] J. FOLEY, A. DAM, S. FEINER, J. HUGHES: *Computer Graphics: Principles and Practice*, 2nd Edition, Addison-Wesley, Reading, Massachusetts, U.S.A., 1990.

[FJ98] L FEIJS, R. D. JONG: *3D Visualization of Software Architectures*. In: Communication of the ACM, Vol. 41, No. 12, S. 73-78, 1998.

[FPF88] K.M. FAIRCHILD, S.E. POLTROCK, G.W. FURNAS*: SemNet: Three-Dimensional Graphic Representations of Large Knowledge Bases*. In: Cognitive Science and its Applications for Human-Computer Interaction. Ed Raymond Guindon Lawrence Erlbaum. 201- 233, 1988.

[FR91] T. M. J. FRUCHTERMAN, E. M. REINGOLD: *Graph Drawing by Force-directed Placement*. In: Software – Practice & Expierience 21(11): 1129-1164, 1991.

[Fur81] G. W. FURNAS: *The fisheye view: a new look at structured files*. Technical Report, Bell Laboratories, Technical Memorandum 81-11221-9, 1981.

[FW94] G. FRANCK, C. WARE: *Representing Nodes and Arcs in 3D Networks*. In: A. L. Ambler, T. D. Kimura: Proceedings IEEE Symposium on Visual Languages, October 4-7, 1994, St. Louis, Missouri. IEEE Computer, Society Press, S. 189-190, 1994.

[GHJ+96] E. GAMMA, R. HELM, R. JOHNSON, J. VLISSIDES: *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Amsterdam, 1996.

[GJS96] J. GOSLING, B. JOY, G. STEELE: *The Java Language Specification*, 1st Edition, Addison-Wesley, Amsterdam, 1996.

[HC98] J. HUNTER, W. CRAWFORD: *Java Servlet Programming*. O'Reilly & Associates Inc, Cambridge u.a., 1998.

[J3D] Homepage zur Java 3D API. URL: http://developer.java.sun.com/developer/earlyAccess/java3D/download.html.

[JDK] *Java Developement Kit 1.1.8*, Copyright Sun Microsystems, Inc., URL:http://java.sun.com/products/jdk/1.1.

[KC98] H. KOIKE, H.-C. CHU: *How Does 3D Visualization Work in Software Engineering? : Empirical Study of a 3D Version/Module Visualization System*. Proc. of Int. Conf. on Softw. Eng., S. 516-519, 1998.

[Koi93] H. KOIKE: *The Role of Another Spatial Dimension in Software Visualization*. In: Transactions on Information Systems, 11(3): 266-286, 1993.

[Küh96] *RALF KÜHNEL: Die Java-Fiebel: Programmierung interaktiver Homepages für das World Wide Web*, Addison-Wesley, Bonn u.a., 1996.

[KY93] H. KOIKE, H. YOSHIHARA: *Fractal approaches for visualizing huge hierarchies*. In: Proceedings of the 1993 IEEE Symposium on Visual Languages, S. 55-60. IEEE/CS, 1993.

[Lea97] D. LEA: *Concurrent Programming in Java(TM) – Entwurfsprinzipien und Muster*, Addision-Wesley, Bonn u.a., 1997.

[Lei99] H. LEISERING: *Neues grosses Wörterbuch – Fremdwörterbuch. Sonderausgabe*, Compact, München, 1999.

[LRP95] J. LAMPING, R.RAO, P. PRIOLLI*: A foucs and context technique based on hyperbolic geometrics for visualizing large hierarchie*. ACM CHI'95, Denver, ACM Press/Addison-Wesley, New York, 401-408, 1995.

[LS87] J. H. LARKIN, HERBERT A. SIMON: *Why a Diagram is (Sometimes) Worth Ten Thousand Words*, Cognitive Science Vol. 11, S. 65-99, 1987.

[Mar97] C. MARRIN: *Proposal for a VRML 2.0 Informative Annex - External Authoring Interface Reference*. Silicon Graphics, URL: http://www.vrml.org/WorkingGroups/vrml-eai/ExternalInterface.html, 1997.

[MM56] MILLER, G. MÄRZ: *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information.* In: The Psychological Review vol. 63(2), S.86, 1956.

[MRS95] B. MONIEN, F. RAMME, H. SALMEN: *A Parallel Simulated Annealing Algorithm for Generating 3D Layouts of Undirected Graphs*. Proc. of Graph Drawing 1995, S. 396-408, 1995.

[NETSCAPE] *Netscape® Communicator 4.6*, Copyright 1994-1999 Netscape Communications Corporation, URL: www.netscape.com.

[JBuilder] *JBuilder Standard Version 2.00*, Copyright 1997-1998 Borland International, Copyright 1999-2000 Inprise Corporation, URL: www.borland.com/jbuilder.

[NVISION] Homepage der Firma *NVision Software Systems Inc*., Fredericton, NB, Kanada, URL: http://www.nvss.nb.ca

[Oes98] *BERND OESTEREICH: Objektorientierte Softwareentwicklung – Analyse und Design mit der Unified Modeling Language*, 4. aktualisierte Auflage, R. Oldenbourg Verlag, München, Wien, 1998.

[*PBS93*] B. A. PRICE, R. M. BAECKER, I. S. SMALL: *A Principled Taxonomy of Software Visualization.* In: Journal of Visual Languages and Computing 4(3) S. 211-266, 1993.

[PFW98] G. PARKER, G. FRANCK, C. WARE*: Visualization of Large Nested Graphs in 3D: Navigation and Interaction.* In: Journal of Visual Languages and Computing, 9, S. 299-317, 1998.

[Phi00] M. PHILIPPSEN: *JavaGrande - Hochleistungsrechnen mit Java*. In: Informatik-Spektrum, 23 (2), S.79-89, 2000.

[RATIONAL] Informationen über Rational Rose lassen sich unter der URL http://www.rational.com/products/rose/index.jtmpl finden.

[RCM93] G. G. ROBERTSON, S. K. CARD, J. D. MACKINLAY: *Information Visualizing using 3D Interactive Animation*, Communications of the ACM, Vol 36. No. 4, 1993.

[Rek93] REKIMOTO, J., GREEN, M.: *The Information Cube: Using Transparency in 3D Information Visualization.* In: Proc. of the Third Annual Workshop on Information Technologies & Systems WITS´93, S. 125-132, URL: ftp://ftp.csl.sony.co.jp:CSL/CSL-Papers/95/SCSL-TR-95-012.ps.gz, Dezember 1993.

[RMC91] G. G. ROBERTSON, J. D. MACKINLAY, S. K. CARD: *Cone Trees: Animated 3D Visualizations of Hierarchical Information*. In: Proceedings of CHI'91. S. 189-194, 1991.

[RMS97] K. RYALL, J. MARKS, S. SHIEBER: *An Interactive Constraint-Based System for Drawing Graphs*. In: Proceedings of UIST'97 Banff, Alberta, Canada, 1997.

[RS99] K. REICHENBERGER, R. STEINMETZ*: Visualisierungen und ihre Rolle in Multimedia-Anwendungen*. In: Informatik-Spektrum, 22 (2), S.88-98, 1999.

[SM93] R.L. SOLLENBERGER, P. MILGRAM: *The effects of Stereoscopic and Rotational Displays in a Three-Dimensional Path-Tracing Task*. Human Factors, 35(3) 483-500, 1993.

[SMM00] *SMM News*. Kundenzeitschrift und Katalog der Firma SMM Software GmbH, Hechtenkaute 5, 55257 Budenheim 5, Ausgabe März 2000.

[Ten94] R. TENNANT: *The 5th Wave*. In: Computer Zeitung, Vol. 25, No. 39,Konradin-V., Leinfelden-Echterdingen, S. 2, September 1994.

[Tol96] I. G. TOLLIS: *Graph Drawing and Information Visualization*. In: ACM Computing Surveys 28(4es), URL: http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a19-tollis, December 1996.

[TS92] J. TESLER, S. STRASNICK: *FSN: The 3D File System Navigator*. Silicon Graphics, Inc. Mountain View, CA, 1992.

[VFAQ98] Frequently Asked Questions (FAQ) der *UseNet*-Gruppe *comp.graphics.visualization*. Wird in der Version vom 6.2.1998 regelmäßig gesendet von Eugene N. Miya (eugene@ marcy.nas.nasa.gov), 1998 (auch auf beigliegender CD-ROM enthalten).

[WC98] K. WALRATH, M. CAMPIONE: *The JFC Swing tutorial – A Guide to Constructing GUIs*, Addison-Wesley, Reading, Massachusetts u.a., 1998.

[WF96] C. WARE, G. FRANCK: *Evaluation stereo and motion cues for visualizing information nets in three dimensions*. In: ACM Transaction on Graphics. 15(2), S. 121-139, 1996.

[WHF93] C. WARE, D. HUI, G. FRANCK: *Visualizing Object Oriented Software in Three Dimensions*. CASCON'93 (IBM Centre for Advanced Studies) Conference Proceedings, S. 612-620, 1993.

[Wün97] V. WÜNSCHE: *Visualisierung strukturierter Informationen mit VRML*. Diplomarbeit, Institut für Computergrafik, Fachbereichinformatik, Universität Rostock, 1997.

[XM92] Y. XIAO, MILGRAM: *Visualization of Large Networks in 3-D Space: Issues in Implementation and Experimental Evaluation*. Proceedings of the 1992 CAS conference. S.247-258, 1992.

[You96] PETER YOUNG: *Three Dimensional Software Visualisation*, Technical Report 12/96, Department of Computer Science, University of Durham, UK, URL: http://www.dur.ac.uk/~dcs3py/pages/work/Documents, Nov. 1996.

[Zie95] *ZIELONKA, R. (GESCHÄFTSFÜHRER): Visual Recall – Hochentwickeltes Dokumenten-Management und Informations-Visualisierung in Netzwerk-Umgebungen,* Xerox XSoft, Düsseldorf, 1995.

[ZK95] R. ZAVODNIK, H. KOPP*: Graphische Datenverarbeitung – Grundzüge und Anwendung,* Hanser-Verlag, München, Wien, 1995.

Anhang A - Inhalt der beiliegenden CD-ROM

Die beiliegende CD-ROM enthält die folgenden Dateien:

- *Install.txt*: Beschreibt die Installation des J3Browsers.
- *Compile.txt*: Erklärt das Kompilieren des J3Browsers.
- *Diagramm.txt*: Beschreibt die nach der Installation des J3Browsers verfügbaren Beispieldiagramme.
- *swingall.jar*: Benötigter Teil der Swing-GUI-Bibliothek.
- *cc32e46.exe*: Installationsprogramm für den Netscape Communicator Version 4.6. Der Communicator wird für den J3Browser benötigt.
- *JBROWSER.ZIP*: Notwendig für die Installation des J3Browsers.
- *VRML.ZIP*: ZIP-Datei mit den Beispieldiagrammen als Exporte im VRML-Format.

Nach der Installation des J3Browsers sind neben dem System selbst zusätzlich noch der Quelltext des Browsers sowie eine mit Javadoc generierte Dokumentation des Quelltexts verfügbar. Auch vom J3Browser lesbare Dateien mit den Beispieldiagrammen sind dann erreichbar (vgl. *Install.txt*).

Anhang B - Das Werkzeug *j3merge*

J3Merge ist ein einfaches Werkzeug, daß dazu dient, verschiedene Diagramme, d.h. Darstellungsdateien, zusammenzuführen. Die kann z.B. dafür genutzt werden, für häufig verwendete Bibliotheken, wie die Java API, einmal ein Diagramm zu erstellen und dies dann ggf. mit weiteren Diagrammen zu verbinden. Gesteuert wird das Werkzeug über die Kommandozeile. Der Aufruf besitzt folgende Syntax:

```
j3merge –help
```

oder

```
j3merge –out neuVis.dar [-sm:neuSM.sm] altVis1.dar ... altVisN.dar
```

Erste Variante führt lediglich zur Ausgabe eines Hilfetexts. Die zweite Variante ist die Wesentliche. Die Darstellungsdateien *altVis1.dar* bis *altVisN.dar* werden dabei zu einer neuen Darstellungsdatei *neuVis.dar* zusammengefügt. Über die Option *-sm* kann ein neues Strukturmodell für die Darstellungsdatei angegeben werden. Wird dies unterlassen, so wird das Strukturmodell aus *altVis1.dar* verwendet.

Das Zusammenfügen verschiedener Darstellungsdateien, die gleichnamige Symbole enthalten, ist nicht möglich. Die verschiedenen Pfeilnegativlisten der Darstellungsdateien werden vereint.

Anhang C - Dokumentanzeige auf Symbolwänden

Die Darstellung von Dokumentation auf Symbolwänden ist wenig zweckmäßig und vergrößert den Szenengraph stark. Deshalb ist sie in der auf der CD-ROM enthaltenen Version der Datei „Protos.wrl“, welche die benötigten VRML-Prototypen definiert, nicht vorgesehen.

Will man sich diese Möglichkeit jedoch einmal anschauen, so ist die Datei zu verändern. Dies geschieht auf einfache Weise dadurch, daß alle Vorkommen der Zeichenfolge „#WANDTEXT“ in der Datei gesucht werden. Diese Vorkommen finden sich immer in einer eigenen Zeile. In der jeweils darauffolgenden Zeile muß das Kommentarzeichen „#“ entfernt werden. Beispiel:

```
#WANDTEXT
#   exposedField MFString wandtext []
```

wird zu:

```
#WANDTEXT
      exposedField MFString wandtext []
```

Gleichzeitig ist beim Laden eines Diagramms die Option „*Dokumente auf Symbolwänden anzeigen*“ zu aktivieren.

Anhang D - Verwendete dreidimensionale Notation

Dieses Anhang liefert eine Übersicht über die gesamte verwendete dreidimensionale Notation.

D.1 Symbole

Die Farbe und Größe von Symbolen kann frei bestimmt werden. Entitätssymbole besitzen auf jeder Seite eine Eigenschaftsmarkierung

Entitätssymbole:



-a- Gewöhnliche Klasse -b- Fehlerklasse -c- Ausnahmeklasse -d- Schnittstelle

Abbildung D.1: Entitätssymbole

Eigenschaftsmarkierung (in der Mitte jeder Seite eines Entitätssymbols):

Zugriffsmodus	Farbe	Beispiel
public	Grün	
protected	Orange	
default	Rot	
private	Grau	

Tabelle D.1: Farbkodierung der Zugriffsmodi

Instanziierbarkeit	Form	Beispiel
abstrakt	Kreis	
konkret	Dreieck	
final	Dioden-zeichen	

Tabelle D.2: Formkodierung der Instanziierbarkeit

Ausführbarkeit durch zwei Rechtecke:



Abbildung D.2: Eigenschaftsmarkierung bei ausführbaren Klassen

Paket- und Paketzusammenfassungssymbol:

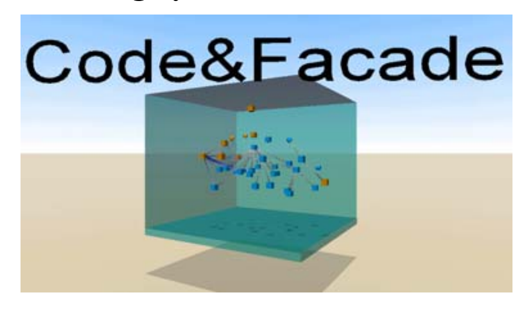
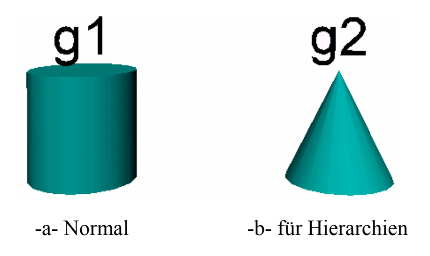


Abbildung D.3: Beispiel für Paketzusammenfassungssymbol (Paketsymbol analog)

Gruppensymbole:



-a- Normal -b- für Hierarchien

Abbildung D.4: Gruppensymbole

Symbolbeschriftung:

Symbolbeschriftungen bestehen aus bis zu zwei Zeilen, wobei die untere – größer geschriebene – Zeile für den Namen und die obere Zeile ggf. für Paketangaben genutzt wird. Je nach Einstellung (vgl. Beschreibung Einstellungsfenster im Abschnitt 10.8) werden in der unteren Zeile einer Symbolbeschriftung anhand eines Präfix und eines Postfix weitere Informationen angezeigt.

<u>*Präfix*</u>: (Aufbau: [ai])

a=A:	Symbol verankert.	*i=T*	Symbol teilweise isoliert.
keine Angabe für a:	Symbol nicht verankert.	*i=V*	Symbol vollständig isoliert.
		keine Angabe für i:	Symbol nicht isoliert.

Wenn ein Symbol weder verankert noch isoliert ist, entfallen die eckigen Klammern.

<u>*Postfix*</u>:

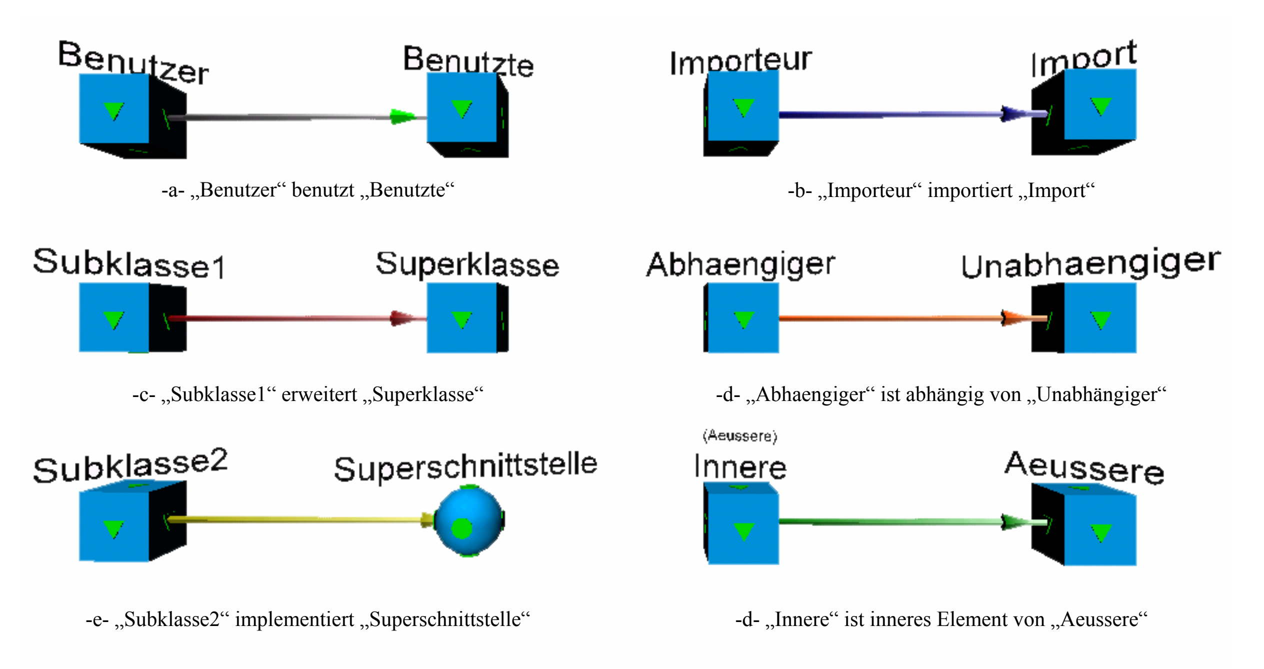
+ bei einem Paketsymbol:	In das Paketsymbol sind Symbole von direkten Subpaketen geschachtelt.
+ bei einem Entitätssymbol:	In das Entitätssymbol sind Symbole von direkten inneren Entitäten geschachtelt.
leer:	keine entsprechende Schachtelung.

<u>*Beispiele*</u>:

[AT]awt+ [V]datatranfer rmi+

D.2 Pfeile

Arten:



-a- „Benutzer“ benutzt „Benutzte“

-b- „Importeur“ importiert „Import“

-c- „Subklasse1“ erweitert „Superklasse“

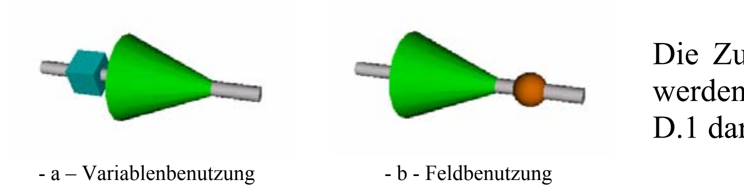
-d- „Abhaengiger“ ist abhängig von „Unabhängiger“

-e- „Subklasse2“ implementiert „Superschnittstelle“

-d- „Innere“ ist inneres Element von „Aeussere“

Abbildung D.5: Verwendete Arten von Pfeilen und ihre Bedeutung

Details von Benutzungen:



- a – Variablenbenutzung

- b - Feldbenutzung

Abbildung D.6: Anzeige der Details einer Benutzung

Die Zugriffsmodi einer Benutzungen werden über die Farben der Tabelle D.1 dargestellt.

Details von Abhängigkeiten:

Abhängigkeitspfeile besitzen an der Spitze Farbmarkierungen, die anhand der Farben für Pfeilarten aus Abbildung D.5 die Arten der vorliegenden Beziehungen angeben.

Details von Erweiterungen:

Erweiterungen mit Redefinitionen können durch eine doppelte Pfeilspitze kenntlich gemacht werden (z.Z. nicht implementiert).